*Proceedings of the Sixth International*

# Workshop on Unique Chips and Systems

# U C A S – 6

**December 4th, 2010**
**Atlanta, GA**

**Held in Conjunction with**
**the 43rd Annual IEEE/ACM International Symposium**
**on Microarchitecture (MICRO-43)**

# U C A S – 6

**December 4th, 2010**
**Atlanta, GA**

# UCAS-6 Organizing Committee

## General Chairs

Byeong Kil Lee, University of Texas at San Antonio
Dhireesha Kudithipudi, Rochester Institute of Technology
Tor Aamodt, University of British Columbia

## Technical Program Committee

Rajeev Balasubramonian, University of Utah
Pradip Bose, IBM TJ Watson
Chen-Yong Cher, IBM TJ Watson
Young Kyu Choi, KUT
Jeanine Cook, New Mexico State University
Manoj Franklin, University of Maryland
Jie Han, University of Alberta
Jaehyuk Huh, KAIST
Ali Irturk, University of California San Diego
Hans Jacobson, IBM TJ Watson
Tejas Karkhanis, IBM TJ Watson
Omer Khan, MIT
Shigeru Kusakabe, Kyushu University
Jeffrey Kuskin, D.E. Shaw Research
Rabi Mahapatra, Texas A&M University, College Station
Saraju Mohanty, Univ. of North Texas
Mike O'Connor, AMD Research
Mark Oskin, University of Washington
Sanghamitra Roy, Utah State University
Karu Sankaralingam, University of Wisconsin
Resit Sendag, University of Rhode Island
Michael Shebanow, NVIDIA
Lei Wang, Univ. of Connecticut

# Table of Content

## Session I: Computer Architecture - 1



## Session II: Computer Architecture - 2

## Session III: VLSI Design

# Session I: Computer Architecture - 1

# Early Experience with Profiling and Optimizing Distributed Shared Cache Performance on Tilera's Tile Processor

Inseok Choi, Minshu Zhao, Xu Yang, and Donald Yeung
*Department of Electrical and Computer Engineering*
*University of Maryland at College Park*
{*inseok,mszhao,yangxu,yeung*}*@umd.edu*

***Abstract*—This paper describes our experience with profiling and optimizing physical locality for the distributed shared cache (DSC) in Tilera's Tile multicore processor. Our approach uses the Tile Processor's hardware performance measurement counters (PMCs) to acquire page-level access pattern profiles. A key problem we address is imprecise PMC interrupts. Our profiling tools use binary analysis to correct for interrupt "skid," thus pinpointing individual memory operations that incur remote DSC slice references and permitting us to sample their access patterns. We use our access pattern profiles to drive *page homing optimizations* for both heap and static data objects. Our experiments show we can improve physical locality for 5 out of 11 SPLASH2 benchmarks running on 32 cores, enabling 32.9%–77.9% of DSC references to target the local DSC slice. To our knowledge, this is the first work to demonstrate page homing optimizations on a real system.**

## I. Introduction

Practically all current high-performance commercial CPUs integrate multiple cores on a single chip. Today, multicore chips with 4-8 cores are commonplace. Several companies have also demonstrated that it is possible to integrate many 10s of cores on-chip [1], while others are shipping manycores [2] that run standard operating systems and are programmable using familiar shared memory models. And since Moore's law scaling will continue at historic rates for the foreseeable future [3], even higher core counts are expected down the road.

A key determiner of multicore performance is the on-chip cache. As the number of cores increases, it becomes necessary to introduce hierarchy or to distribute the cache across the chip and provide independent access to separate cache banks in order to keep up with the on-chip parallelism. A multicore in which the shared cache is distributed among the processor's cores is called a distributed shared cache (DSC [4]) architecture. Memory references to such physically distributed shared caches exhibit non-uniform cost since data placed in a cache bank close to a requesting core can be accessed more quickly than data placed in a distant bank. Even when the caches are coherent, the cache misses will exhibit a non-uniform cost. This can affect performance each time the distributed cache is accessed–*i.e.*, when a miss occurs from a cache higher up in the on-chip memory hierarchy.

On processors with distributed caches, higher performance can potentially be achieved by managing on-chip physical locality so that data are placed in the cache banks closest to their referencing cores. Such bank *homing optimizations* can be controlled either in hardware at cache-block granularity or in software at page granularity. Hardware techniques, which are implemented within the cache coherence protocol, typically map the cache blocks on different banks based on memory block addresses. Software techniques typically rely on the operating system to provide homing information via the virtual memory layer, thus enabling individual pages to be homed on different banks. Page-based techniques often require profiling to determine per-page access patterns for driving the page homing decisions. Apart from homing optimizations, it is also possible to replicate and/or migrate data at runtime to further improve physical locality (*e.g.*, to track dynamically changing access patterns).

Several researchers have explored homing optimizations in the past, with significant prior work related to both hardware cache block-based [5], [6], [7], [8], [9], [10], [11] as well as software page-based [12], [13], [14], [15] techniques. However, all of this prior research was conducted on simulators. To our knowledge, no study has applied homing optimizations on real processors. Such research is important because it can highlight real-world issues overlooked by simulation studies that must be addressed before possible benefits can be realized.

In the past, processors did not implement distributed caches, so real-system studies were not possible. But this is no longer the case today. For example, Tilera Corporation has recently shipped many-core CPUs that use a tiled CMP architecture. In these *Tile Processors* [2], the lowest level of cache employs a cache-coherent distributed shared cache architecture. A typical Tile processor DSC is composed of 64 independent cache "slices" distributed amongst the cores, hardware maintains cache coherency and operating system provide homing information. When a processor makes a given memory reference and suffers a cache miss, the cache coherency mechanism directs the miss to a *home cache* on another core on the chip, thereby potentially averting a costly off-chip DRAM access. The coherency hardware then moves the referenced data automatically to the referencing core's

cache so that subsequent references may be satisfied locally. In this architecture, cache misses incur a variable cache access latency, making homing optimizations relevant.

This paper presents our early experience with improving physical locality in a Tile Processor's DSC. Our work focuses on how to apply page-based homing optimizations on the Tile CPU, making the following contributions. First, we present a novel technique for acquiring fine-grain page-level access pattern information for driving page placement decisions. Although only information about which threads access which pages is needed, determining this requires pinpointing individual memory instructions so that the memory addresses, and hence pages, each thread accesses can be profiled. Our solution leverages the Tile Processor's hardware performance measurement counters (PMCs) to sample the effective addresses of individual memory instructions. PMCs enable low-overhead profiling, but they are typically not designed to provide per-instruction sample resolution. A key part of our solution is to use binary analysis to correct the imprecise hardware samples, thus pinpointing individual memory instructions that reference remote slices and permitting us to profile their access patterns.

Second, we use our access pattern profiles to drive homing decisions, to place the pages on the tile that accesses them the most. Specifically, we try to explicitly home and improve physical locality for pages in the heap and static data memory regions. We currently only optimize pages that are referenced primarily by a single core, placing them on the DSC slice closest to the core with the most references to the page. Our optimizations do not allow page migration. Instead, placement decisions made at memory allocation time are fixed for the duration of the program's run.

Finally, we conduct experiments using programs from the SPLASH2 benchmark suite [16] that demonstrate our profiling and optimization techniques. Our results show we can improve physical locality for 5 out of 11 SPLASH2 benchmarks running on 32 cores, enabling 39.3%–77.9% of DSC references to target the local DSC slice. Moreover, we find our homing optimizations already exploit most of the potential physical locality in the SPLASH2 benchmarks. Significant improvements can only come by creating more opportunities for homing, perhaps by addressing false sharing via smaller virtual memory pages.

The remainder of the paper is organized as follows. Section II presents our access pattern profiling techniques. Then, Section III describes how we home pages based on the access pattern profiles. Next, Section IV discusses our experiments. Finally, Section V concludes the paper.

## II. Access Pattern Profiles

Software page-based techniques require access pattern information to drive page homing decisions. In particular, the distribution of references performed by cores on a per-page basis is needed. In previous work, such access pattern profiles were obtained via simulation which is slow and requires architectural simulators. To enable page-based techniques on real systems, it is crucial to develop more efficient techniques. This section describes how access pattern profiles can be acquired using hardware PMCs. Section II-A begins with an overview of the Tile Processor architecture and its PMC support. Then, Section II-B discusses the problem of profiling individual memory instructions, and describes how we address the problem. Finally, Section II-C presents the profiling system we built.

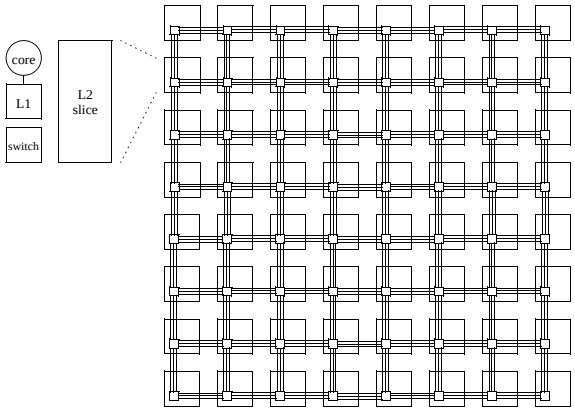



Figure 1. A typical Tile Processor is composed of 64 tiles, each containing a VLIW core + L1 cache, an L2 cache "slice," and an on-chip network switch.

### A. Tile Processor

A typical Tile Processor, illustrated in Figure 1, consists of a grid of 64 general-purpose VLIW cores and interconnected by multiple 2D mesh on-chip networks. Each core has its own private split L1 cache, and a local L2 cache that acts as one slice of a distributed shared cache. The core and its associated cache are connected to the on-chip networks through a switch. The switch, core, and cache are referred to as a *tile*. Cores can access their local L2 slice with minimal latency, but incur increasingly higher latencies to access more distant L2 slices due to inter-tile communication across the switched interconnect.

Tile Processors allow several ways in which data can be placed across the DSC caches, including on a page-by-page basis in which each page can be homed on any given core. This permits flexible OS-controlled distribution of data. Since our work focuses on page-based homing, we use the Tile Processor's per-page mechanism exclusively.

In the per-page approach, every virtual memory page is assigned its own home tile. The home tile's L2 cache is where cache blocks from the page are cached on-chip. In Section III, we will discuss how software can specify the home for each page, thus controlling data placement on-chip.

To enable measurement of low-level hardware events, the Tile Processor supports 2 32-bit hardware performance measurement counters per tile. Each hardware PMC can

observe one of 99 pre-defined hardware events at any moment in time. These events monitor instruction execution in the cores, memory operations in the memory hierarchy, as well as traffic across the on-chip network. The Tile Processor runs a Linux operating system which supports OProfile, a UNIX system-level utility for accessing the hardware PMCs. In addition, we ported PAPI [17] and Perfmon2 [18] to the Tile Processor.[1] These are standard APIs that export a fuller set of PMC features to users compared to OProfile.

### B. Using PMCs to Profile Memory References

For every page in memory, we profile the number of references each core makes to the page in the DSC, thus identifying the most frequently referencing core(s) on a per-page basis at the DSC level. We only profile read references (loads) since these are the main source of performance degradation. (Stores write to a store buffer on a cache miss. They do stall when a memory fence is performed, but the programs we study, *e.g.* SPLASH2, are coarse grain parallel programs in which fences are very infrequent. Therefore, stores rarely stall in such programs).

The Tile Processor's PMCs can monitor a remote-read hardware event which is useful for acquiring access pattern profiles. A remote-read event occurs each time a core issues a load instruction that misses in the local L1 cache and then hits in a remote L2 slice. This monitors all DSC references except for those issued by the core on the referenced page's home. To get around this problem, during profile runs, we home all pages on a spare tile not running any of the compute threads, thus forcing all DSC references to be non-local and allowing them to be monitored by the remote-read event.

While the PMCs can count DSC references, they alone cannot associate the counts to pages and cores. For this, we rely on *sampling*. Hardware PMCs can be configured to deliver an interrupt after a pre-set number of remote-read events have occurred, allowing an interrupt handler to periodically sample load references to the DSC. In particular, each interrupt can identify the core performing the load, as well as the particular load instruction involved (*i.e.*, its program counter or PC). Moreover, given knowledge of the particular load being sampled, the interrupt handler can probe the register containing the load's effective address and identify the referenced page. In this fashion, each interrupt/sample can attribute a single page reference to a particular core. After a large number of such samples, we can determine *statistically* the frequency with which all pages in a program are referenced by each core.

One obstacle to implementing this approach is the Tile Processor's PMCs (as well as those on most other commercial CPUs) does not provide per-instruction sampling resolution. The problem is PMC interrupts are not precise.

[1]The latest versions of PAPI are implemented on top of Perfmon2.

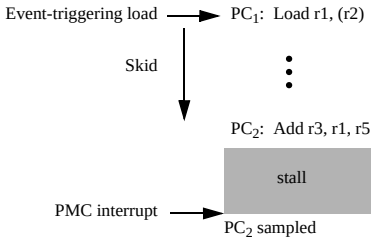



Figure 2. Imprecise handling of PMC interrupts on the Tile Processor results in sampling of the instruction dependent upon the event-triggering load ($PC_2$) rather than the load itself ($PC_1$).

When a PMC interrupt is signaled, the core keeps executing. At some later time, the interrupt is actually serviced, but by then the core may have executed past the event-triggering instruction. If so, the PC sampled is not the load performing the DSC reference, but rather some other PC further down the instruction stream. Such PMC sampling "skid" is not a problem when trying to locate the function or thread incurring an event, but it prevents pinpointing individual memory instructions which is necessary to profile their access patterns.

Fortunately, it is possible to correct for sampling skid on the Tile Processor due to certain features of its pipeline. The Tile CPU employs a register file with presence bits [19] that allow execution past cache-missing loads, providing some latency tolerance. Rather than the cache-missing load stalling the pipeline, the first instruction to use the load's target register stalls, as illustrated in Figure 2.[2] In practice, we find the delay in signaling a PMC interrupt is larger than the def-use distance for loads that reference the DSC (we observe a def-to-use of 1–20 VLIW instruction bundles), but smaller than the latency for the remote L2 slice access. Hence, the PMC interrupt almost always samples the instruction *dependent* on the event-triggering load.

By performing dependence analysis, we can identify the event-triggering load instruction from the sampled PCs: it is the first load preceding the sampled PC whose destination register matches one of the sampled instruction's source registers. Usually, we encounter the event-triggering load in the same basic block as the sampled instruction. However, in some cases, the event-triggering load resides in the basic block preceding the block containing the sampled instruction. To handle these cases, we perform dependence analysis across basic blocks when necessary.

### C. Profiling Tools

We perform two profiling runs to acquire the access pattern profiles. The first addresses the imprecise PMC interrupt problem described in Section II-B. It collects all of the imprecisely sampled PCs that occur in the profiled program. Then, after this profiling run completes, we perform binary

[2]It is possible that the first use does not stall if the cache-miss latency is completely overlapped, but this is not the common case.

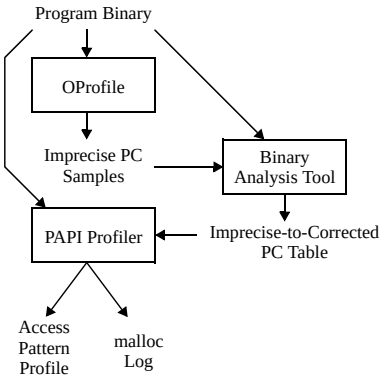



Figure 3. Profiling infrastructure for acquiring access pattern profiles.

analysis to correct the sampling skid and identify the event-triggering loads. From this analysis, we build a table that associates the imprecise PCs with their corresponding corrected PCs, along with the register containing the effective address of the event-triggering load at the corrected PC.

The second profiling run acquires the actual access pattern profiles. During this profiling run, each sampling interrupt consults the imprecise-to-corrected PC table computed from the first profiling run to identify the load responsible for the interrupt as well as its effective address register. As discussed in Section II-B, the interrupt handler probes the register to determine the referenced page, and logs the sample (core ID and page number) in a profile table. At the end of the second profiling run, this profile table–which contains the access pattern profile–is output to the user.

Figure 3 illustrates the tools involved in profiling. We use the OProfile utility (see Section II-A) in its unmodified form to collect the imprecise PC samples. We built our own binary analysis tool to construct the imprecise-to-corrected PC table. This binary analyzer extracts a control flow graph from the program binary to permit inter-basic block analysis when searching for the corrected PCs. Finally, we use PAPI to acquire the access pattern profiles. We modified PAPI to download the imprecise-to-corrected PC table into the kernel. We also modified PAPI's kernel-level PMC counter overflow handler to perform the PC sample correction and load effective address identification.

In addition to profiling access patterns, we also log all calls to malloc, the heap memory allocator. During each malloc call, we record the call site as well as the dynamic instance for that call site (in case it is executed multiple times). When each malloc call returns, we record the starting address and size of the allocated object. This information allows us to associate pages in the access pattern profiles back to individual heap objects, and to identify where (call site and dynamic call instance) those objects were created. As the next section will show, this information can be used for optimizing heap objects.

## III. Page Homing Optimization

Once the access pattern profile and malloc log have been acquired for a given program, subsequent executions of the program can use them to drive page homing optimizations. This section presents our optimizations. First, Section III-A describes the access patterns that we target. Then, Sections III-B and III-C explain how we drive page homing for the heap and static data regions, respectively.

### A. Optimization Opportunities

Our page homing optimization tries to home pages residing in the heap and static data memory regions on the tiles where they are referenced most frequently. Currently, our optimization targets pages in the access pattern profiles that are referenced primarily by a *single core*. Figure 4 shows an example access pattern profile, illustrating the different access patterns and objects we optimize.

In Figure 4, we graph the access pattern profile for a 16-core execution of Ocean, a program from the SPLASH2 benchmark suite [16]. Pages are plotted along the X-axis while cores are plotted along the Y-axis. The graph plots the normalized number of samples acquired for each page from each core along the "Z-axis" (extending out of the paper). Samples that are particularly large are highlighted by the shaded peaks. As Figure 4 shows, the pages numbered 106 to 883 are referenced primarily by a single core (*i.e.*, at each X-axis point in this range, there is always a single Y-axis point with a dominant peak). These are the pages our optimization tries to explicitly home.

In addition to identifying the pages to optimize, we must also identify which program-level objects the pages belong to. This is particularly important for heap objects because it determines which malloc calls must be instrumented to control homing (see Section III-B). In practice, we find there are two different types of objects. The first is illustrated in Figure 4 by pages 148–274 and 274–442 which form diagonal access patterns that increase in core ID with increasing page number. Each of these two memory regions is a single object (in this example, they are both on the heap and each is allocated by a single malloc call). Due to their diagonal access pattern, each object is accessed by all the cores, but most of the per-core accesses are destined to mutually exclusive and contiguous pages in the object. These two memory regions are examples of *distributed arrays*. They can be optimized by distributing their pages in chunks across neighboring tiles to match their diagonal access patterns.

The second type of object is illustrated in Figure 4 by pages 106–127 and 442–883 which form diagonal access patterns that decrease in core ID with increasing page number. Again, most of the per-core accesses in these two memory regions are destined to mutually exclusive and contiguous pages. But instead of one object containing all of the pages on the diagonal, each set of pages that are referenced by the same core is a separate object (*i.e.*, on the

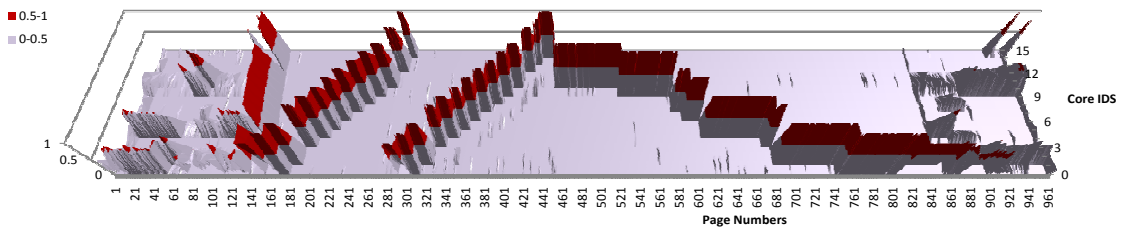



Figure 4. Example access pattern profile of a 16-core execution of Ocean from the SPLASH2 benchmark suite. Page numbers are plotted along the X-axis while core IDs are plotted along the Y-axis. Normalized sample count per core/page is plotted along the "Z-axis" (extending out of the paper).

heap, each would be allocated by a separate malloc call). These memory regions are examples of *privately accessed objects*. They can be optimized by homing all of their pages on the tile where most of the memory references occur.

The remaining pages in Figure 4 in the ranges 1–106 and 883–950 are primarily accessed by multiple cores, usually 2 or 3. Although not shown in Figure 4, another common case is pages that are accessed equally by all the cores. Our optimization does not try to improve physical locality for such shared pages. Instead, we simply distribute shared pages in round-robin fashion across tiles.

### B. Homing Heap Pages

Page homing in the heap can be controlled via the Tile Processor's mspace abstraction. A standard Linux parameter, mspace is a segment, with a particular homing policy for all pages in the segment. By default, the heap resides in a single mspace that homes its pages on the tile performing the first malloc to each page. For programs in which core 0 allocates all of the heap objects (*i.e.*, most of the SPLASH2 programs), this default policy places the entire heap on tile 0.

To improve physical locality for the different heap objects and access patterns described in Section III-A, we create multiple mspaces with different homing policies. We also provide a custom malloc function in a separate optimization library that can select between these different mspaces, thus binding different homing policies to heap objects as they are allocated at runtime. Users need only link their program against our optimization library, and provide the access pattern profile and malloc log for their program to enable our heap optimizations.

For privately accessed heap objects, our optimization library creates one mspace per tile, with each mspace homing its pages on a unique tile. At allocation time, the custom malloc function selects one of these mspaces according to the access pattern profile, thus homing the entire object on the tile where most of its references occur.

For heap-based distributed arrays, our optimization library creates an mspace that distributes pages across tiles so that each portion of the distributed array resides in its referencing core's local tile. To achieve the desired physical locality, we set the distribution *chunking factor–i.e.*, the number of contiguous pages to place on one tile before moving onto the next tile–to be the ratio of the distributed array size and the number of tiles in the machine times the page size.[3] Since each mspace can only support a single chunking factor, we must create one mspace for every unique chunking factor across all of the distributed arrays in the program.

In order to select the appropriate mspace for each allocated heap object, our custom malloc function consults the malloc log and access pattern profile acquired during the profiling runs. In particular, as the custom malloc function is called at runtime, it matches the call to its corresponding call of malloc in the malloc log. (The custom malloc function keeps track of the same call site and dynamic call instance information logged during profiling, as described in Section II-C, to enable matching). Once the corresponding malloc call from the profiling run is identified, the heap object being allocated can be determined along with its access pattern. If the heap object is a distributed array or a privately accessed object, then the custom malloc function allocates the object onto the mspace that supports the object's access pattern. Otherwise, the custom malloc function allocates the object onto a default mspace that distributes the object's pages across tiles in round-robin fashion.

Since our optimizations are profile-driven, their effectiveness is sensitive to discrepancies in access patterns between the profiling and optimized runs. The chunk size that each thread accesses will be different if the input data size changes. In particular, it may be desirable for optimized runs to use a different input problem or core count compared to the profile runs. Our optimization library tries to compensate for changes to these two parameters. For example, our custom malloc function adjusts the chunking factor for distributed arrays if array size and/or machine size changes from profiling run to optimized run. However, aside from problem input and core count variation, we do not compensate for any other factors that may alter access patterns at runtime, for example dynamic work distribution (*e.g.*, using work queues).

Our current page homing optimizations for the heap are mostly (though not fully) automatic. As mentioned above, users must link their programs against our optimization

[3]The chunking factor may not be an integral number of pages. Mspaces permit specifying a separate chunking factor per tile in the distribution. This allows placement of the majority of a distributed array's elements on the optimal tile.

| Benchmark | Input | Benchmark | Input |
|---|---|---|---|
| FFT | $2^{20}$ points | Ocean | 1026 grid |
| Barnes | 16384 bodies | Water-NS | 1000 molecules |
| Cholesky | tk17.O | Water-SP | 1000 molecules |
| Radix | 2097152 keys | Radiosity | 7832 objects |
| LU | 1024 matrix | Raytrace | ball4 |
| FMM | input.2048 | | |

Table I
SPLASH2 BENCHMARKS USED IN OUR STUDY ALONG WITH THEIR INPUT PROBLEM SIZES.

library. In addition, they must call our library initialization routines which requires adding 4 lines of code to their program. Aside from this, there are no additional source code changes needed to apply our heap optimizations.

### C. Homing Static Data Pages

Unlike heap objects, static data objects are allocated at compile time, and are bound to a particular mspace. Hence, they are already assigned a home by the time a program begins execution. Similar to the heap, the default policy is to home all pages from the static data region on tile 0.

To control page homing in the static data region, we use memory mapping and unmapping to change the homing policy from the default policy. In particular, we identify all pages in the static data region from the access pattern profile that are referenced primarily by a single core. Next, we copy the contents of these identified pages to an external file. Then, we unmap the copied pages from the program's address space, and map into their place the copied data from the external file using the `mmap_mbind()` system call. Similar to mspaces, the mmap_mbind() system call permits specifying a home tile for the mapped pages. Hence, this permits per-page homing control.

In our current implementation, we determine the pages to optimize in the static data region manually, and insert the unmapping and mapping calls manually into the program source code. However, due to the systematic nature of these analyses and source code instrumentation, we believe it is possible to automate them in the future.

## IV. Experimental Results

This section demonstrates the profiling and optimization techniques discussed in Sections II and III, and studies the potential benefits they can provide. In particular, our experiments quantify the number of remote L2 slice references that are converted into local L2 slice references by the page homing optimizations. We begin by discussing experimental methodology in Section IV-A. Then, Section IV-B presents our results.

### A. Experimental Methodology

We conducted all experiments on a Tile Processor running the Linux operating system from the Tilera MDE version 2.1. To drive our study, we use the entire SPLASH2 benchmark suite [16] except for volrend. We used `tile-cc` (the Tile Processor's C compiler) to compile the benchmarks with the highest level of optimization. Table I lists the benchmarks and the input problems we used in the experiments.

Unfortunately, we encountered some bugs in our page homing code that prevented us from running with a large number of cores. At the time of writing this paper, we were unable to perform profiling and optimized runs on more than 32 cores for a number of SPLASH2 benchmarks. So, we only report experiments on at most 32 cores.

For each benchmark binary, we acquire access pattern profiles and malloc logs using the profiling tools described in Section II-C. All of our profiles are acquired on 32-core executions of the benchmarks. Then, we instrument the benchmark source codes to call our optimization library initialization routines and to perform the homing optimizations for the static data region, as discussed in Sections III-B and III-C. Lastly, we re-compile the benchmarks, linking them against our optimization library, and run them to measure optimized performance. These optimized runs use the same configurations as the profiling runs.

To quantify improvements, we compare the optimized and unoptimized benchmarks. As discussed in Sections III-B and III-C, the default homing policy places all pages in the original unoptimized benchmarks on tile 0. To provide a better baseline against which to compare our techniques, we link the unoptimized benchmarks against our optimization library, but configure the system to distribute all heap and static data pages across tiles with a chunking factor of 1. This utilizes the DSC capacity fully, but randomly distributes pages across the on-chip L2 slices.

In our results, we report sampled page references at the DSC level. Since we use a sampling frequency of 7000, sampling counts can be converted into page reference counts (at least approximately) by multiplying by 7000. (The selection of a proper sampling frequency is important. If the sampling frequency is too small, profiling will incur large overhead; but if the sampling frequency is too large, less frequent events may not be sampled. Our choice of 7000 for the sampling frequency was determined experimentally, and works well for SPLASH2 benchmarks. It may be necessary to tune this sampling frequency parameter for other benchmarks.) Lastly, we only report measurements in the parallel region of each benchmark. We exclude program initialization, which is performed at the beginning of each SPLASH2 benchmark on a single core.

### B. Physical Locality Results

Table II reports our page reference count results. In particular, the $2^{nd}$ and $3^{rd}$ columns of Table II (labeled "Total") report the number of sampled page references in each benchmark's profiling run that are destined to the heap and static data memory regions, respectively. This data shows that across our benchmarks, heap objects receive more memory references than objects in the static data region,

| | Total | | Baseline | | | Optimized | | | Potential | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Heap | Static | Heap | Static | % Total | Heap | Static | % Total | Heap | Static |
| FFT | 5376 | 8387 | 289 | 536 | 6.0% | 5372 | 536 | 42.9% | 5376 | 0 |
| Barnes | 8197 | 11324 | 521 | 711 | 6.3% | 521 | 7152 | 39.3% | 246 | 6920 |
| Cholesky | 37361 | 6735 | 1890 | 389 | 5.2% | 1907 | 389 | 5.2% | 113 | 2 |
| Radix | 4299 | 79 | 276 | 5 | 6.4% | 3404 | 5 | 77.9% | 3425 | 15 |
| LU | 0 | 2 | 0 | 0 | 0% | 0 | 0 | 0% | 0 | 2 |
| FMM | 19667 | 123 | 1583 | 9 | 8.0% | 1583 | 9 | 8.0% | 19667 | 1 |
| Ocean | 90703 | 26030 | 5400 | 1387 | 5.8% | 87857 | 1387 | 76.5% | 88783 | 0 |
| Water-NS | 543 | 3211 | 22 | 190 | 5.6% | 438 | 190 | 16.7% | 543 | 0 |
| Water-SP | 415 | 0 | 1 | 0 | 0.2% | 1 | 0 | 0.2% | 415 | 0 |
| Radiosity | 4741 | 1824 | 430 | 68 | 7.6% | 430 | 68 | 7.6% | 192 | 259 |
| Raytrace | 30750 | 14580 | 1796 | 964 | 6.1% | 1796 | 964 | 6.1% | 5 | 0 |

Table II
NUMBER OF SAMPLED PAGE REFERENCES TO THE HEAP AND STATIC DATA REGIONS IN TOTAL, THAT ARE DESTINED TO LOCAL L2 SLICES IN THE BASELINE AND OPTIMIZED BENCHMARKS, AND THAT CAN BE POTENTIALLY OPTIMIZED.

but both types of objects are important. (One case with anomalous behavior is LU which we will discuss shortly).

The $4^{th}$ and $5^{th}$ columns of Table II (labeled "Baseline") report the number of sampled page references in the unoptimized benchmarks that are destined to local L2 slices broken down into heap and static data references, respectively. The $6^{th}$ column of Table II reports the percentage of the total sampled references that these baseline local references represent–*i.e.* $(\%\ Total)_{Baseline} = \frac{(Heap+Static)_{Baseline}}{(Heap+Static)_{Total}} \times 100$. This data shows the unoptimized benchmarks exhibit poor physical locality. Only 5%–8% of all DSC references are to local L2 slices. In other words, more than 90% of DSC references must traverse the on-chip network to communicate with a remote L2 slice. This makes sense because page homing in the unoptimized benchmarks is essentially randomized across the Tile Processor's DSC.

The $7^{th}$ and $8^{th}$ columns of Table II (labeled "Optimized") report the number of sampled page references in the optimized benchmarks that are destined to local L2 slices broken down into heap and static data references, respectively. The $9^{th}$ column of Table II reports the percentage of the total sampled references that these optimized local references represent–*i.e.* $(\%\ Total)_{Optimized} = \frac{(Heap+Static)_{Optimized}}{(Heap+Static)_{Total}} \times 100$. As this data shows, our page homing optimizations improve physical locality for 5 benchmarks: FFT, Barnes, Radix, Ocean, and Water-NS. In these benchmarks, 39.3%–77.9% of DSC references are to local L2 slices, a 6–12X increase over the baseline. For the remaining 6 benchmarks, our homing optimizations do not find many pages to optimize (*i.e.*, that are referenced primarily by a single core), so the number of localized DSC references does not change compared to the baseline.

The remaining columns in Table II provide insight into how much of the potential physical locality in our benchmarks we actually exploit. Since our homing optimization must place each page on a specific tile, it is only effective for pages that are referenced by a small number of cores. In particular, pages that are shared by most/all of the cores in the machine are unlikely to yield any benefit. The $10^{th}$ and $11^{th}$ columns of Table II (labeled "Potential") report the number of samples destined to pages in the heap and static data regions, respectively, that are referenced by *no more than half the cores* (*i.e.*, 16 cores) in the profiling runs. Although some of these pages can still be "widely shared," we believe these sampled reference counts are a good estimate for the potential physical locality improvement.

Comparing the "Potential" and "Optimized" results in Table II, we see our optimizations capture most of the physical locality in the SPLASH2 benchmarks–*i.e.*, many of the optimized heap and static data counts are close to the corresponding potential heap and static data counts. (In some cases, the optimized counts are actually larger than the potential counts. These are due to references destined to local L2 slices for pages shared by more than half the machine.) The greatest missed potential is in FMM where there are a large number of heap references none of which are optimized. There is also some missed potential in Water-SP. But overall, our homing optimizations are fairly comprehensive.

These results suggest that for our optimizations to do substantially better, we must create more opportunities for homing. Comparing the last two columns against the $2^{nd}$ and $3^{rd}$ columns of Table II, we see there is a significant discrepancy between the potential and total sampled reference counts, especially for pages in the static data region. This implies there are a large number of references to pages shared by most of the machine. Upon closer examination, we found a major reason for this is false sharing induced by the Tile Processor's large page size, 64 KB. We believe our optimizations can become more effective if page size is reduced.[4] For pages with false sharing, a smaller page size can create more pages with low-degree sharing that our optimizations can exploit.

Finally, Table II shows LU cannot be optimized because it does not exhibit any sampled references. This is due to the fact that LU performs function calls very frequently. The calls are so frequent that the interrupt handler skid after a remote-read event almost always straddles a function call (*i.e.*, all interrupts sample the called code). Unfortunately, our current binary analysis tool cannot analyze across func-

[4]In fact, the Tile Processor's TLBs can support smaller pages, but the current OS doesn't exploit this hardware feature.

tions, so we fail to identify any of the event-triggering loads in LU. We verified by hand that LU does indeed present significant opportunities for our homing optimizations. In the future, we plan to support inter-procedure analysis in our binary analysis to handle cases like LU.

## V. Conclusions

This paper describes our experience with page-level homing optimizations on a real system, Tilera's Tile Processor running a Linux OS. We show hardware PMCs can be used to acquire page-level access pattern profiles. Moreover, we show that binary analysis can be used to correct for interrupt skid–due to imprecise PMC interrupts–to pinpoint individual memory operations incurring remote-core references and sample their access patterns. We find our page homing optimizations driven by our access pattern profiles can improve physical locality for 5 out of 11 SPLASH2 benchmarks, enabling 39.3%–77.9% of DSC references to target the local L2 slice. In addition, we find our homing optimizations already exploit most of the potential physical locality in the SPLASH2 benchmarks. Significant improvements can only come by creating more opportunities for homing, perhaps by addressing false sharing via smaller virtual memory pages.

## References


[1] Y. Hoskote, S. Vangal, N. Borkar, and S. Borkar, "Teraflop Prototype Processor with 80 Cores," in *Proc. of the Symp. on High Performance Chips*, 2007.

[2] http://tilera.com/products/processors, "Processors from Tilera Corporation."

[3] "Silicon Industry Association Technology Roadmap," 2009.

[4] A. Agarwal, "Tiled Multicore Processors: The Four Stages of Reality," http://groups.csail.mit.edu/cag/raw/documents/tiled-processors-ieee-micro-keynote-2007.pdf 2007.

[5] B. M. Beckman and D. A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," in *Proc. of the 37th Int'l Symp. on Microarchitecture*, Portland, OR, December 2004, pp. 319–330.

[6] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. of the 33rd Int'l Symp. on Comp. Arch.*, June 2006.

[7] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," in *Proc. of the 32nd Int'l Symp. on Comp. Arch.*, Madison, WI, June 2005.

[8] Z. Guz, I. Keidar, A. Kolodny, and U. C. Weiser, "Utilizing Shared Data in Chip Multiprocessors with the Nahalal Arch." in *Proc. of the Int'l Symp. on Parallelism in Algorithms and Arch.*, Munich, Germany, June 2008.

[9] E. Herrero, J. Gonzalez, and R. Canal, "Distributed Cooperative Caching," in *Proc. of the Int'l Conf. on Parallel Arch. and Compilation Techniques*, Toronto, Canada, October 2008.

[10] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," in *Proc. of the Int'l Conf. on Supercomputing*, Boston, MA, June 2005.

[11] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. of the 32nd Int'l Symp. on Comp. Arch.*, Madison, WI, June 2005.

[12] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *Proc. of the 39th Int'l Symp. on Microarchitecture*, December 2006.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proc. of the Int'l Symp. on Comp. Arch.*, Austin, TX, June 2009, pp. 184–195.

[14] L. Jin and S. Cho, "SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors," in *Proc. of the 18th Int'l Conf. on Parallel Arch. and Compilation Techniques*, Raleigh, NC, September 2009.

[15] L. Jin, H. Lee, and S. Cho, "A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors," in *Proc. of the 2006 ACM SIGPLAN Workshop on Memory System Performance and Correctness*, October 2006.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. of the 22nd Int'l Symp. on Comp. Arch.*, Santa Margherita Ligure, Italy, June 1995.

[17] "Performance Application Programming Interface."

[18] "The Hardware-Based Performance Monitoring Interface for Linux."

[19] T.-F. Chen and J.-L. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," University of Washington, 92-06 03, June 1992.

# A Formalized Task Migration Framework for Multiple Configurable Processors Shared Memory SoC Platforms

Hao Shen Frédéric Pétrot
System Level Synthesis Group, TIMA Laboratory,
CNRS/Grenoble INP/UJF
46, Avenue Félix Viallet, 38031, Grenoble, France
{hao.shen@imag.fr frederic.petrot@imag.fr}

***Abstract*—In the quest for increasing and ever changing functionalities, it is expected that the next generation Systems-on-Chips (SoCs) will embed several clusters of CPUs sharing the same memory, as flexibility is more easily realized through software. However, specialized computation engines, such as ASIC and DSP, have a much better Mips per Watt ratio [11] at the price of no or much less flexibility. In order to try to benefit from both flexibility and performance per Watt, one approach consists of building clusters of configurable processors [24], *i.e.* processors that share a common instruction set but can be individually customized (usually with DSP like instructions) to accelerate a given computation. Yet, such a cluster of some processors may be underutilized because of their instruction set specificities. In order to be able to benefit from the available computational power by load balancing, we propose and formalize a task migration framework for shared memory multiple configurable processors platforms. With proposed load balancing algorithms, we experimentally show doubled performance/cost improvements on a shared memory cluster that includes 3 configurable processors.**

## I. Introduction

Consumer electronics devices, such as cell phones, portable media-players, high definition TVs, etc, now require a computational power that largely exceeds the abilities of the most advanced embedded uniprocessor. To satisfy these requirements, the solution of choice in the recent years has been to include different kinds of processing units, MCUs, GPP, DSPs, *Application Specific Instruction-set Processors* (ASIPs) and so on, in the SoCs. In this approach, each processing unit is dedicated to the kind of computation it performs well. These heterogeneous platforms, that feature high dedicated performance, acceptable programmability and good performance per Watt, are currently the choice of the industry [22], [4]. However, next generation SoCs will need to be more versatile, (due to the cost of masks, fast adaptation to changing standards, and expected capability to support multiple applications), and thus will require to be more easily programmable. In order to benefit from application level parallelism and thread level parallelism, mainly for power efficiency and area (thus yield and again power) optimization reasons, one foreseeable solution is the use of highly parallel clustered architectures, even for consumer electronic applications [12].

Assuming identical processors, it has long been known that it may be more efficient in shared-memory multiprocessor cluster to schedule a task on one processor instead of an other [9]. Since inevitably some processors will be idle in a cluster (because of work load changes and non-uniformity of task deadlines), a trade-off between keeping the workload balanced among processors and scheduling tasks where they run most efficiently has to be found [28]. The definition of a load balancing strategy implicitly relies on the capability to migrate one task from one processor to another. However, to benefit from instruction level parallelism at a low power cost, each cluster could be based on a set of application specific Configurable Processors.

From the realization point of view, many ASIP solutions follow the strategy of having a core instruction set and be extensible with application specific instructions. These extensions can either be built automatically by profiling and extracting from the software tasks [1], [20] or be developed manually with compiler supports [1]. The target of these extensions is to improve *Instruction Level Parallelism* (ILP). Also the *Thread Level Parallelism* (TLP) can be satisfied with multiprocessors [15]. These ASIP based MPSoC platforms have both parallelism advantages [14] and may become a powerful and energy efficient solution.

As both heterogeneity and workload balance ability are essential for future MPSoC platforms, we formalize in this paper a framework which can support both configurable processing units and task migration ability. Because the key constraint of the task migration is that the system software, such as OS and drivers, should be able to execute on all underlying processors, we assume that all processors share the same core instruction set and register file for the system software. Beside this, instructions and registers related to computations can be totally different for each processor, in order to provide acceleration for different applications. In contract to the heterogeneous MPSoC architecture composed with multiple isolated SMP subsystems in Fig. 1 (a), Fig. 1 (b) describes a platform in which all ASIPs share the same memory space which includes the same OS image and tasks. Therefore tasks can migrate between different kinds of processors at no other cost than

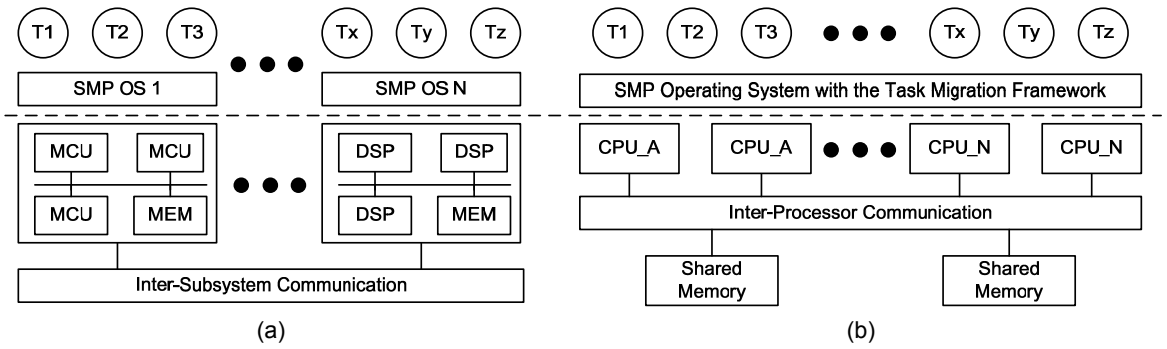



Fig. 1. Heterogeneous MPSoC Software/Hardware Architectures. (a): the traditional MPSoC architecture based on several isolated subsystem. (b): the architecture based on multiple different configurable processors with the shared memory and coherent caches.

increasing cache misses. As an additional benefit, the task migration capability provides some support for fault tolerance, and this is an important issue with the increasing process variability.

The rest of this paper is organized as follows: section 2 discusses the related works. In section 3, we formalize the problem with both hardware and software definition for our heterogeneous task migration framework. Based on the formal definitions, we get the compatibilities of processors and tasks which are used for several load balancing algorithms. The details of migration realization and of the load balancing algorithms are given in section 4. Section 5 presents experimental results that demonstrates the feasibility of the approach and its advantages compared to fully homogeneous and fully heterogeneous platforms. At last, section 6 concludes this paper and future works.

## II. Related Work

Computation migration frameworks have been heavily researched in the past for parallel and distributed computers. Smith [26] gives a survey as of 1988 of process migration, defined as the way of *transferring* the relevant part of the state of a process in order to be able to continue its execution on an other processor. Ignoring the low level performance issues, Smith considers the problem trivial for shared memory machines. Taking an opposite view, Squillante and Lazowska [27] advocate the clever use of the affinity of a task for a processor, in order to avoid trashing the cache while still allowing migration to balance the workload. Among others, these early works set the bases for process and task migration strategies.

With the advance in integration, the migration of tasks has become again an interesting topic for SMP MPSoC architectures. Since small processors have small caches, task migration using the first-come first served approach was considered a good enough solution [25]. Later, Bertozzi et *al.* [8] proposed a task migration framework for an integrated system that uses processor local storage for the task data. Their solution requires the explicit copy of data at identified checkpoints, as in distributed systems, even though the platform features caches and access to shared data. Although homogeneous architectures are more simple to understand and take decision for, they cannot benefit from differences in application requirements. It is currently accepted that the pure SMP solution cannot yet reach the power/performance budget of the integrated systems of the consumer markets, that thus still relies on more *ad-hoc* solutions [19].

Some recent works have focused on load balancing through task migration on heterogeneous platforms. Beltrán et *al.* [7] consider this option theoretically while Nollet et *al.* [21] focus on some feasibility aspects. These approaches rely on the definition of checkpoints for which a *transferable* sub-state of the system is known, but do not deal with simple but major problems such as differences in endianness or word length.

As there is a huge complexity gap between homogeneous and heterogeneous MPSoC, some researchers try to provide a trade-off solutions which can take advantages of both. R. Kumar et *al.* [16], M. Becchi et *al.* [6], S. Balakrishnan et *al.* [5] and S. Ghiasi et *al.* [13] suggest to integrate several processors which implement the same instruction set but with different costs and performances. This limited heterogeneity can achieve higher performance than strict SMP with similar costs. Based on this platform, some scheduling algorithms [10] are designed to achieve higher performance with less power consumption.

The approach we propose shares the same idea but breaks the uniform instruction set constraint of pure SMP systems. ASIP, introduced almost 30 years ago [30], is now a viable solution. One kind of ASIPs is based on a set of well selected basic core instructions and registers. Designers can improve the performance by adding user-defined extended instructions. Commercial products such as Xtensa [1] and CoWare [2] are able to produce efficient configurable processors of that kind, along with stable cross-compilation chains. Our previous work [25] follows this trend and targets task migration ability based on this kind of platforms. There is a similar work for general-purpose computers from [18] which also realizes a heterogeneous architecture task migration framework relaying on the processor exception mechanism. Different from this work, ours is based on predefined extension requirements before binary compilation. For lack of formalization work [25],

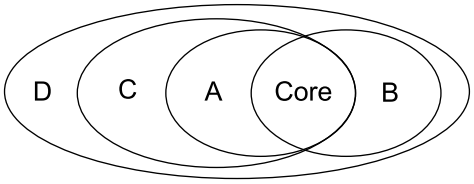



Fig. 2. Instruction Set Relationship

[18], to best of our knowledge, this paper is the first work which formalizes a task migration mechanism based on this kind of heterogeneous shared memory MPSoC architectures.

## III. Hardware and Software Definition

In this section, we define formally the instruction set relationship and explain our formalism using a simple example. We also define a compatibility relationship between tasks and processors of one heterogeneous MPSoC platform based on this formalism.

### A. Instruction set relationship

To explain the relationship between processor instruction sets and the task migration ability, we use the concise example of Fig. 2. This diagram represents the instruction set relationship of one configurable heterogeneous MPSoC platform. In this platform, we have 5 different kinds of instruction sets which are **Core**, **A**, **B**, **C** and **D**. We assume that the instruction set relationship is the same as the register file relationship and therefore we have a single relationship. In following parts, we discuss the *Core Instruction Set* and *Extended Instruction Set* separately.

*1) Core Instruction Set:* As our work is based on configurable processors, a strong prerequisite is that all processors share the same core instruction set shown in the center of Fig. 2. Because we need to realize our task migration framework based on this core instruction set, we define the following groups of instructions.

- **Arithmetic Logic Instructions**: which are used for arithmetic and logic operations.
- **Memory Access Instructions**: which are used for transferring data between memories and registers.
- **Program Flow Control Instructions**: which are used for changing the program execution flow based on the processor status.
- **Concurrent Access Control Instructions**: which are used to serialize requests and avoid non-coherent shared memory access cases in multi-processors execution environments.

Besides these classes of instructions, we also need the following 4 groups of core register files for operating system realization.

- **General Purpose Registers**: which are used for the storage of the data and address information.
- **Program Counter Registers**: which are used to indicate the current program address.
- **Program Status Registers**: which are used to store current processor statuses and include the exception status, the interrupt status and so on.
- **Program Stack Register**: which is used for the current stack address and can be realized using one of the general purpose registers.

Due to the generality of operating system software, it is possible and efficient to build a real operating system in which tasks can migrate among multiple heterogeneously extended processors by using only the core instruction set and the core register file. To ensure this stringent requirement, the designers should not remove critical components (such as atomic instruction, interrupt masking and handling) used by this core instruction set and register file during the processor configuration process.

*2) Extended Instruction Set:* As we use extended instructions to benefit from the data and instruction level parallelism of the applications, we define extended instruction sets and the set relationship between them. Most extended instructions can be divided into either SIMD or MIMD instructions, both requiring independence between the parallel computations.

Besides extended instructions, it is also often necessary to extend the register files to improve the performance.

- **Very Wide Registers**: to improve SIMD instructions performance, we add very wide register files to store the instruction operands and results.
- **Special Internal Registers**: which are used for some special operations such as accumulator registers for the multiply-accumulate operation.

Normally, these extended instructions can be chosen by the application programmer based on the benchmark profiling results and the existing architecture and area/power constraints. Automatically extracting the candidate extended instruction set from one specific application is also the focus of researches by both academia and industry for a long time, see for example the book by Leupers [17]. Among others, recent proposals are L. Pozzi et *al.* [23], F. Sun et *al.* [29] and Xtensa Processor Extension Synthesis (XPRES) Compiler [1] are such works. The formalism we propose does not make any assumption on how the extended instructions are extracted, and thus the resulting ISA can be classified accordingly.

As both read and write of user extended registers are crucial for the context switch and related detailed migration operations, we should also make sure that the extended instruction sets include specific load and store instructions to access all these extended registers. In the context related functions, the extended register files used or required by the task should be load or stored properly by using this kind of extended instructions. For example, we have the ARM [3] processor with the NEON coprocessor extension for media related applications. The NEON extension integrates thirty-two 64 bit double word registers which can only be load and stored by using NEON extended instructions such as `VLDn` and `VSTn`.

*3) Formal Definition:* In one task migration enabled heterogeneous $P$ processors MPSoC platform, we have $N \leqslant P$ different instruction sets that all include the core instruction set $I_{core}$ and the core register file $R_{core}$. We call $S_{core} = I_{core} \cup R_{core}$ the union of the core instruction set and core register set. Meanwhile, we note the extended instruction set as $EI_i$ and the extended register file as $ER_i$.

**Definition 1**: Sets Relationship.

- For all processors in one heterogeneous MPSoC platform, we have the instruction set $I_i$ and register file $R_i$ definition:

$$\forall 1 \leqslant i \leqslant N : I_i = EI_i \cup I_{core}$$

$$\forall 1 \leqslant i \leqslant N : R_i = ER_i \cup R_{core}$$

- For all processors in one heterogeneous MPSoC platform, we have the extended instruction and register set $ES_i$ definition:

$$\forall 1 \leqslant i \leqslant N : ES_i = EI_i \cup ER_i$$

- For all processors in one heterogeneous MPSoC platform, we have the instruction and register set $S_i$ definition:

$$\forall 1 \leqslant i \leqslant N : S_i = I_i \cup R_i = S_{core} \cup ES_i$$

- For all instruction sets in one heterogeneous MPSoC platform, we have the instruction set group $\mathbb{S}$ definition:

$$\mathbb{S} = \{S_1, S_2, ..., S_N\} = \{S_i : 1 \leqslant i \leqslant N\}$$

With these definitions, we can easily express the relationship of Fig. 2. We note the instruction set group $\mathbb{S} = \{S_{core}, S_A, S_B, S_C, S_D\}$. Because the core instruction set is one part of each processor, we have the requirement that for $P$ processors in one heterogeneous MPSoC platform: $\forall 1 \leqslant i \leqslant N, S_{core} \subseteq S_i$. Thus, for the special case of Fig. 2, if all of $ES_i$ are not empty, we have following expressions: $S_A \subset S_C$, $S_C \subset S_D$ and $S_B \subset S_D$.

### *B. Task compilation and migration*

As extended instructions are only efficient for some specific applications, or even more precisely for some kernels of an application, we compile some application tasks and threads with the basic instruction set while some others with extended ones. The basic threads can be migrated for execution on any available processors while extended application tasks and threads can only be executed on processors which should realize this instruction set extension. In the example of Fig. 2, as the instruction set $S_A$ is a subset of $S_C$ and $S_D$, it means that $S_A$ has less extended instructions than $S_C$ and $S_D$. On one side, if some tasks compiled to $S_C$ instruction set and use instructions which do not belong to $S_A$, they are not able to be migrated to processors only support the $S_A$ type instruction set. On the other side, if some tasks just use the $S_A$ instruction set, it is no problem for them to run on $S_C$ and $S_D$ types processors. This execution relationship is presented in Fig. 3.

The tasks, heterogeneously extended processors and execution relationship can be represented as a the **Compatibility Graph** $\mathcal{G}$ shown in Fig. 3.

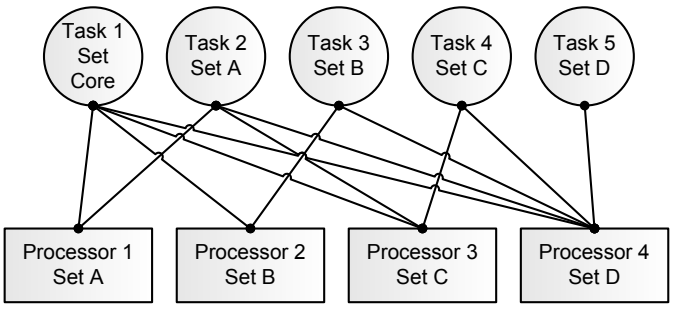



Fig. 3. Processors and Tasks Compatibility. All tasks $T_i$ and processors $P_j$ realize one of $S_k$ from the total instruction set $\mathbb{S}$. The execution relationship is represented with connection between tasks and processors.

**Definition 2**: Migration possibility of the heterogeneous MPSoC $\mathbb{M}$ is defined as $\mathbb{M} = (\mathbb{S}, \mathbb{T}, \mathbb{P}, \mathbb{G})$.

- $\mathbb{S}$ is the set that includes all instruction sets used in one heterogeneous MPSoC platform (the same as Definition 1).
- $\mathbb{T}$ represents the task set which include $N_T$ tasks for one application system and $N_T \geqslant 1$. We have $\mathbb{T} = \{T_1, T_2, ...T_{NT}\}$. When task $T_i$ is compiled onto one specific instruction set $S_j$, we can represent the *is compiled for ISA* relationship with the symbol $\star$. For this case, we have $T_i \star S_j$.
- $\mathbb{P}$ is the set of processors which includes $N_P$ different processors for one MPSoC platform and $N_P \geqslant 1$. When processor $P_i$ realizes one specific instruction set $S_j$, we can present the *realizes ISA* relationship with the symbol $\triangle$. For this case, we have $P_i \triangle S_j$.
- A **Bipartite Compatibility Graph** $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$ represents the compatibility relationship between each $T_i \in \mathbb{T}$ and $P_j \in \mathbb{P}$. An edge $\{T_i, P_j\} = c_k \in \mathbb{C} \subseteq \mathbb{T} \times \mathbb{P}$. This edge $c_k$ means task $T_i$ can be executed by processor $P_j$.

$$\forall i, j, T_i \in \mathbb{T}, P_j \in \mathbb{P}$$

$$(T_i \star S_k) \wedge (P_j \triangle S_l) : S_k \subseteq S_l \Longleftrightarrow c(T_i, P_j)$$

With both Def. 1 and Def. 2, we clarify the compatibility relationship among instruction sets, tasks and processors. In Fig. 3, we have an example of compatibility with 4 processors and 5 tasks. So we have the task set $\mathbb{T} = \{T_1, T_2, T_3, T_4, T_5\}$ and the relationship between tasks and instruction sets $\{T_1 \star S_{core}, T_2 \star S_A, T_3 \star S_B, T_4 \star S_C, T_5 \star S_D\}$. Meanwhile, we have the processor set $\mathbb{P} = \{P_1, P_2, P_3, P_4\}$ and the relationship between processors and instruction sets $\{P_1 \triangle S_A, P_2 \triangle S_B, P_3 \triangle S_C, P_4 \triangle S_D\}$. For the task compatibility of this example, we have the compatibility relationship: $\mathbb{C} = \{c(T_1, P_1), c(T_1, P_2), c(T_1, P_3), c(T_1, P_4), c(T_2, P_1), c(T_2, P_2), c(T_2, P_3), c(T_2, P_4), c(T_3, P_2), c(T_3, P_3), c(T_3, P_4), c(T_4, P_3), c(T_4, P_4), c(T_5, P_4)\}$. All these instruction sets, tasks, processors and compatibility relationships are represented by the compatibility graph $\mathcal{G}$. From this simple example, we can clearly find tasks compiled with the

core instruction set have the best flexibility while the tasks compiled with extended instruction sets can only be executed by specific processors.

## IV. Heterogeneous Task Scheduling Algorithms and Realization

As the compatibility of tasks and processors is readily visible with our task migration framework, we now need to provide a way to choose the right process and thread to migrate or elect. This is the goal of our heterogeneous task scheduling algorithms. Based on the instruction set compatibility rules, we adapt several existing task scheduling algorithms to our heterogeneous MPSoC task migration framework. Because different configured processors can execute different classes of tasks, our task scheduling algorithms try to utilize the extended instruction set advantage and trade-offs between the scheduler efficiency and the execution efficiency. The formal descriptions and realization details of these scheduling algorithms are also given in this section.

### A. Task scheduling algorithms

With both Def. 1 and Def. 2, for a specific processor $P_i$, compatible tasks can be grouped into the set $\mathbb{T}_i$ where $\mathbb{T}_i = \{T_{i1}, T_{i2}, ..., T_{ij}\}$. Based on the *compatibility graph* $\mathcal{G}$, there should be an edge between each $P_i$ and $T_{ij}$ to guarantee the compatibility. In the following algorithm descriptions, both $P_i$ and $\mathbb{T}_i$ are used to present this compatibility property.

Beside the compatibility property, we also need a value to evaluate the efficiency of processor computation. For a task $T_i$ running on a processor $P_j$, we have the difference $D(T_i, P_j)$ defined as:

$$\exists S_i, S_j \in \mathbb{S}, (T_i \star S_i) \wedge (P_j \triangle S_j) : D(T_i, P_j) = |S_j - S_i|$$

This definition represents the distance between the instructions and registers that the processor $P_j$ provides and the $T_i$ task requires. The bigger number of $D(T_i, P_j)$ means the more unused instructions provided by processor $P_j$ which wastes computation ability and power. Two of the following scheduling algorithms are designed to take account of this efficiency problem. During the scheduling process, $\mathbb{T}_{queue}$ is used to defined all tasks inside the runnable queue structure. Meanwhile, $|\mathbb{T}_{queue}|$ is defined as the size of this task queue.

*1) FMFS algorithm: First Match First Serve* **(FMFS)** is one of the simplest algorithms for our heterogeneous MPSoC platform. The basic idea is just add the compatible constraints into the traditional FIFO like scheduling algorithms. When a processor is ready for new tasks execution, it goes through the task queue and picks up the first compatible task to execute. Though this algorithm is simple and efficient for the scheduler realization, it does not fully optimize for extended instruction sets of heterogeneous processors. With this FMFS algorithm, tasks with smaller instruction sets generally have better execution chances which decrease the whole system performance.

**Algorithm**:
Search the queue in order to select the first task that is compatible.
**Performance**:
Complexity of this algorithm is $O(1)$.

For implementation, we have following abstract code.

**Input**:
A compatibility graph $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$.
A free processor $P_i$ and a task queue $\mathbb{T}_{queue}$ in FIFO order.
**Output**:
If exist, select a compatible task for the processor $P_i$.
**Implementation**:

```
1  for j = 1 to |T_queue| in FIFO order
2     if c(T_j, P_i) ∈ C // T_j is compatible with P_i
3        T_j is the result and finish this scheduling process
4     end if
5  end for
6  Default idle task is the result // No compatible tasks
```

*2) Most compatible algorithm:* To fully take the advantage of powerful extended instruction sets, we define the most compatible algorithm. By using this algorithm, when a processor is ready for new tasks execution, it iterates over the whole task queue and compares the CPU instruction set with each waiting task. After the whole task queue is checked, the compatible task which uses the most instructions is chosen for execution. As this algorithm emphasizes tasks using extended instruction sets, in some cases, it may provide better overall performance. But we should also notice that tasks compiled only with the core instruction set may be in a starving situation if tasks making use of extension are always ready to run.

**Algorithm**:
Search the queue and execute the most compatible task.
**Performance**:
Complexity of this algorithm is $O(|\mathbb{T}_{queue}|)$.

For implementation, we have following abstract code.

**Input**:
A compatibility graph $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$.
A free processor $P_i$ and a task queue $\mathbb{T}_{queue}$.
**Output**:
If exist, select a compatible task for the processor $P_i$.
**Implementation**:

```
1  A empty candidate task set T_candidate = φ.
2  forall T_j ∈ T_queue
3     if c(T_j, P_i) ∈ C // T_j is compatible with P_i
         //Add T_j to candidate set
4        T_candidate = T_candidate ∪ T_j
5     end if
6  end forall
7  if T_candidate ≠ φ
8     choose the first (or any) task from the set T:
      T = min(D(T_j ∈ T_candidate, P_i)).
9  else Default idle task is the result // No compatible tasks
10 end if
```

*3) Priority based most compatible algorithm:* To avoid the drawbacks of both the FMFS algorithm and the most compatible algorithm, we combine these two algorithms together and create the priority based most compatible algorithm. In this algorithm, we add a priority level to each task. Instead of having a single task queue, we have several task queues, each corresponding to a priority. This allows to limit the search time and avoids the starving situation, by increasing the priority of long waiting threads. Meanwhile, for each priority level, we still use the most compatible algorithm to find the best candidate for one processor. The priority level for each task can be adjusted depending on some priority calculation algorithms to avoid starving situation and decrease the overall system response time. The drawback of this algorithm is the complex realization and the scheduler performance heavily depends on priority setting and adjustment algorithms.

**Algorithm**:
Search from the highest priority queue and execute the best
compatible waiting task.
**Performance**:
Complexity of this algorithm is $O(|\mathbb{T}_{queue}|)$.

For implementation, we have following abstract code.

**Input**:
A compatibility graph $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$.
A free processor $P_i$.
Multiple task queues $\{\mathbb{T}_{queue\_1}, \mathbb{T}_{queue\_2}, ..., \mathbb{T}_{queue\_n}\}$ for
$n$ different priorities.
**Output**:
If exist, select a compatible task for the processor $P_i$.
**Implementation**:

```
1  A empty candidate task set T_candidate = φ.
2  for k = 1 to n // n queues with different priorities
3     forall T_j ∈ T_queue_k
4        if c(T_j, P_i) ∈ C // T_j is compatible with P_i
           // Add T_j to the candidate set
5          T_candidate = T_candidate ∪ T_j
6        end if
7     end forall
8     if T_candidate ≠ φ
9        choose the first (or any) task from the set T:
         T = min(D(T_j ∈ T_candidate, P_i)).
10    end if
11 end for
12 Default idle task is the result // No compatible tasks
```

### B. Scheduler realization

To realize all discussed task migration algorithms on one heterogeneous MPSoC platform, we should well identify instruction sets provided by processors and used by tasks. Besides this, we also introduce three some non-realtime task migration. The target of these algorithms is just to show the advantages of the taks migration framework.

*1) Instruction Set Identification:* As the instruction set representation is important for both processors and tasks, the scheduler of the operating system should have a special mechanism to store this information. We have `ISA_ID` to represent the instruction set information. Then we assign one `CPU_ISA_ID` for each processor and one `TASK_ISA_ID` for each task. The use of specific ID to indicate processor instruction set differences is a method commonly used in industry. As the instruction set relationship is complex in our platform, we would like to have the `ISA_ID` to well represent the instruction set relationship. By using this ID, it is convenient for the scheduler to handle the relationship in run-time environments.

In our framework, we have instruction sets $\mathbb{S} = \{S_1, S_2, ..., S_{NS}\}$ and the relationship $\mathbb{R} = \{S_1 \supseteq S_2, ..., S_{NS-1} \supseteq S_{NS}\}$. We map the instruction set to the natural number set the $\mathbb{N} = \{1, 2, ...\}$ and the relationship to *Bit OR* relationship.

In Fig. 2, we have 4 different extended instruction sets and the core instruction set. In Table. I, we assigned each separated instruction group with a *bit* and each `ISA_ID` with a binary number with bit validation for each small instruction group. By using `ISA_ID`, we replace the complex instruction set relationship with simple bitwise operations. For each processor, the `CPU_ISA_ID` is the same as the `ISA_ID` of the one realized. Meanwhile, for each task, the `Task_ISA_ID` is the instruction set `ISA_ID` used by the compiled binary. The compatibility relationship between `CPU_ISA_ID` and `TASK_ISA_ID` is also illustrated in Table. I.

*2) Instruction Set Based Scheduler Realization:* With the definition of both `CPU_ISA_ID` and `TASK_ISA_ID`, we use the bitwise *or* operation to handle the instruction set compatibility test. To test compatibility, we need only this operation:

```
(CPU_ISA_ID | TASK_ISA_ID) == CPU_ISA_ID
```

This test only relies on simple bit and compare operations to make the computation efficient for the frequent usage $c(T_j, P_i) \in \mathbb{C}$ in all heterogeneous task scheduling algorithms.

*3) CPU_ISA_ID and TASK_ISA_ID Integration:* In our task migration framework, each processor should have a `CPU_ISA_ID` which presents the instruction set and register file it realizes. In realization, we add one specific read-only register to each processor which is hard coded to indicate the corresponding `CPU_ISA_ID`. The task migration framework should access this register during all task operations.

The `TASK_ISA_ID` is assigned to each task during its creation. We have modified the POSIX Thread standard `pthread_attr_t` structure by adding the `TASK_ISA_ID` tag. When a task is created with standard `pthread_create`, this ID is transfered to the OS kernel and used for the heterogeneous task scheduling described before and the context related functions realization in the following discussion.

*4) Context Related Functions Realization:* The context related functions for heterogeneous MPSoC platforms are

TABLE I
PROCESSORS AND SUPPORTED TASKS ISA IDENTIFICATION EXAMPLE.

| Set | CPU_ISA_ID | Compatible Task | Compatible Task_ISA_ID |
|---|---|---|---|
| Core | 0x0000 | Core | 0x0000 |
| A | 0x0001 | Core and A | 0x0000, 0x0001 |
| B | 0x0010 | Core and B | 0x0000, 0x0010 |
| C | 0x0101 | Core, A and C | 0x0000, 0x0001, 0x0101 |
| D | 0x1111 | Core, A, B, C and D | 0x0000, 0x0001, 0x0010, 0x0101, 0x1111 |

more complex than for homogeneous ones. There are different extended register files $R_i = ER_i \cup R_{core}$ in each processor $i$, so we need to handle these extra registers $ER_i$ in our context related functions. In contract to loading and storing all extended registers during these operations, we should only touch the one used by the previous executed task and required by the next executed task. For example, the new context switch function should include following four steps:

- Store all core registers to the stack of the previous executed task.
- Store the necessary extended registers depending on the `TASK_ISA_ID` of the previous executed task.
- Load all core registers from the stack of the next executed task.
- Load all necessary extended register based on the `TASK_ISA_ID` of the next executed task.

As a processor may run a task that makes use of only a subset of the extended register, we save the registers depending on the instruction set used by this task. This save action is feasible because the registers used by the task is a subset of the processor registers and the load action is also feasible for the same reason. The context structure has a shared part that contains the $R_{core}$ registers, and a private part that depends on the extended registers used by the task $ER_i$.

Though the realization of the context related functions make the structure of each task context different depending on the `TASK_ISA_ID`, it can avoid much unnecessary stack memory occupation and save the time of register operations. We still take the ARM [3] processor example with the NEON coprocessor extension. As the NEON extension integrates thirty-two 64 bit double word registers, it should consume minimal 256 bytes context memory space. Besides the memory cost, loading and storing all these extra registers wastes some time and power which are important for embedded systems. With our task migration framework, we can fix the context size of the task dependent on its `TASK_ISA_ID` when created. If the task does not use the NEON extended instructions and registers, the extra memory and operation time are saved with our task migration framework.

## V. EXPERIMENTAL RESULTS

In this section, we introduce the Motion-JPEG example to present the performance, cost and power advantages of our task migration framework for on heterogeneous MPSoC platforms based on the same core instructions.

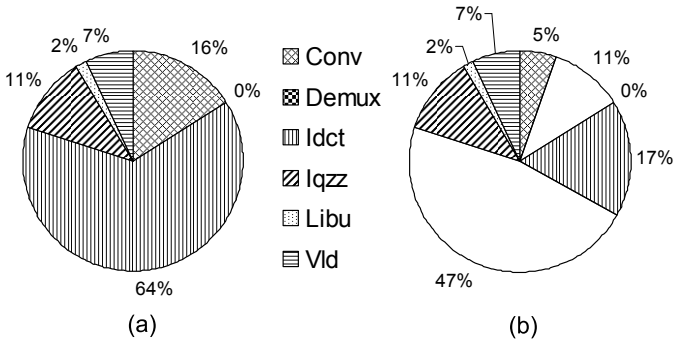



Fig. 5. Performance difference. (a)Computation time with only the core instruction set. (b) Computation time with both core and extended instruction set.

### A. Introduction of the Motion-JPEG case study

The *Motion-JPEG* is a multimedia format in which a video sequence is separately compressed as JPEG images. In this case study, we realize the Motion-JPEG decoder application with eight initial tasks: TG, DEMUX, VLD, IQZZ, IDCT, CONV, LIBU and RAMDAC (Fig. 4). It works by reading a stream of JPEG images with the Traffic Generator (TG) task and writing the decoded pixels into the Random Access Memory Digital-to-Analog Convert (RAMDAC).

Mutek [25] is a lightweight SMP POSIX Thread compliant operating system kernel. To adapt it to our heterogeneous platform, the scheduler part of this kernel is modified for heterogeneous MPSoC task migration requirements.

### B. Heterogeneous MPSoC Architecture

Beside system software and application software, we use a heterogeneous MPSoC architecture. In this architecture, all configured processors use the basic instructions of the Xtensa LX2 processor. The extended instruction set compiler and the software compiler are also provided by Tensilica [1].

After the task profiling of the Motion-JPEG example, we get the left part of Fig. 5 which indicates the computation time used for each tasks based on the core instruction set. As we focus on the application optimization, this figure only shows the 6 software tasks, the operating system and communication costs are excluded to make the comparison clearer. In this figure, we find the IDCT and the YUV to RGB Converter (CONV) are two most time consuming tasks which consume respectively 64% and 16% of CPU time. The

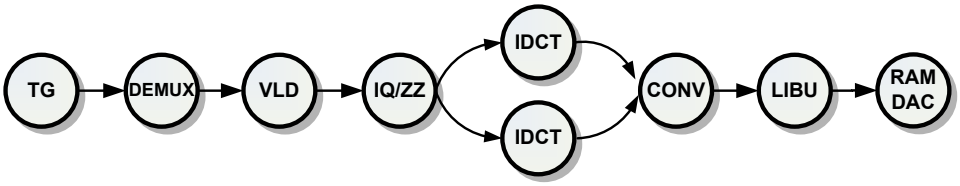



Fig. 4. Functional Model of the Motion-JPEG Case Study

TABLE II
INSTRUCTION SET AND REGISTER FILE EXTENSION

| | IDCT Core | CONV Core |
|---|---|---|
| New Instructions | 10 | 14 |
| New Registers | 4 | 28 |
| Extra Gates | 70,904 | 63,869 |
| Total Gates | 139,904 | 132,869 |
| Speedup Effect | 380% | 309% |

following instruction set extension work focuses on these two tasks.

Table. II shows user defined extended instruction sets for both IDCT and CONV tasks. With extended instructions, IDCT and CONV tasks speed up more than 3 times and the new computation times of these 6 tasks is shown at the right of Fig. 5. In this Figure, the circle represents 100% of the original CPU load, and the white parts ($47\% + 11\% = 58\%$) represent the gain in CPU load. It is obvious to identify the efficiency of extended instructions for real applications. With this table, we also give the hardware cost of these extended instructions and registers. This information is useful to show the advantage of our task migration framework from a Cost/Performance ratio perspective.

Cache is also an important component of one MPSoC platform. As different cache organizations and sizes can change the system performance, cache can also become a big part of heterogeneous platform configuration. In our experiment, all processors have only instruction caches which are direct mapping 4KB cache of 256 blocks of 4 words. All data access are uncached, as the simulation platform we use does not support cache coherence. Because instruction cache can still show the effect of task migration, experiment results we get do not impact our experiment target which is to show the task migration advantage of heterogeneous MPSoC.

Our heterogeneous MPSoC architecture that makes use of the extended Xtensa processors and that is utilized in the following experiments is depicted in Fig. 6. Based on this architecture, we have two different task migration frameworks. The traditional one is to port OS separately for each processor and there are no task migration and scheduling between different processors. The fixed mapping solution is shown in Fig. 6. Our task migration framework is to let all processors share the same OS image and make task migration possible among different extended processors.

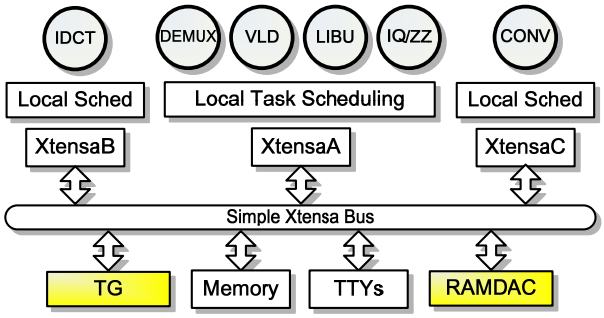



Fig. 6. Motion-JPEG Case Study. This system includes three heterogeneous processors based on the same Xtensa LX2 core instruction set. Processor A is just the basic processor without any extended instructions. Processor B and C include extended instructions for IDCT and CONV separately.

### C. Performance and Cost Advantage

As we know task migration can take advantage of CPU idle time, we use the Motion-JPEG to show, using the same heterogeneous MPSoC architecture, the difference in execution efficiency between fixed task mapping and dynamic task migration.

The results of Table. III compare the performance of four different scheduling solutions. Three of them use the architecture presented in Fig. 6, but with different scheduling algorithms. Because of flexibility, performance of both heterogeneous scheduling algorithms overcomes that of the traditional fixed task assignment framework. From this table, we can easily find the performance advantage of the two task scheduling and migration algorithms that we propose (almost 100% higher performance). We also show that different scheduling algorithms have different performance results. For this particular case, the FMFS algorithm has better performance than the most compatible algorithm because it has a simple and efficient realization.

In contrast to these heterogeneous architectures, in the last column, we show the SMP architecture in which each processor includes all extended instructions and registers. Compared with the heterogeneous architecture, the SMP architecture is flexible and high performance. But we should also notice that the hardware cost of this SMP architecture is much higher than the heterogeneous one. For the performance/cost ratio (we normalize the data in Table. III), the heterogeneous architecture with our task migration framework is much better than the SMP one. With this case study, the migration algorithms we propose are better utilizing the extended instruction set and thus achieve higher performance. This result shall not be

TABLE III
COMPARISON OF IDLE TIME FOR DIFFERENT SCHEDULING FRAMEWORKS BASED ON THE SAME HETEROGENEOUS ARCHITECTURE

| | Fixed Task Assignment | FMFS Algorithm | Most Compatible Algorithm | SMP Task Scheduling |
|---|---|---|---|---|
| Frames/second | 1.44 | 2.88 | 2.70 | 2.88 |
| Gates number | 341,773 | 341,773 | 341,773 | 611,295 |
| Normalized Perferformance/Cost | 0.50 | 1.00 | 0.94 | 0.56 |

generalized to other examples without caution.

## VI. CONCLUSIONS AND FUTURE WORKS

This paper has formalized a task migration framework based on the configurable heterogeneous MPSoC architecture. With the Motion-JPEG example, we show the performance/cost advantage of our framework over existing SMP architectures and fixed task mapping framework. Meanwhile we should also notice that though the heterogeneity property can help accelerate overall system performance, a large percentage of application tasks only rely on the core instruction set to be well scheduled among all execution units. Our formalized task migration framework can make the heterogeneous architecture much more flexible for application designers than the traditional hard mapping one. In the near future, we can improve performance analyse by using more complex benchmarks.

## REFERENCES


[1] Tensilica Inc., Xtensa Microprocessor, 2009. [Online]. Available: http://www.tensilica.com.

[2] CoWare Processor Designer, 2009. [Online]. Available: http://www.coware.com/products/processordesigner.php.

[3] ARM Inc., ARM Series Processor, 2009. [Online]. Available: http://www.arm.com.

[4] J. Augusto de Oliveira. Nexperia computing architecture for connected consumer applications. In *In Proceedings of the 20th International Conference on VLSI Design*, page 31, Jan. 2007.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA*, pages 506–517. IEEE Computer Society, 2005.

[6] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Conf. Computing Frontiers*, pages 29–40. ACM, 2006.

[7] M. Beltrán, A. Guzmán, and J. L. Bosque. Dealing with heterogeneity in load balancing algorithms. In *Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 123–132, Timisoara, Romania, july 2006.

[8] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe*, pages 15–20, 2006.

[9] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *Software Engineering, IEEE Transactions on*, 5(4):341–349, July 1979.

[10] J. M. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. H. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 101–112, Washington, DC, USA, 2007. IEEE Computer Society.

[11] H. De Man. Ambient intelligence: gigascale dreams and nanoscale realities. pages 29–35, Feb. 2005.

[12] E. Flamand. Strategic directions towards multicore application specific computing. In *Proceedings of the 2009 Design, Automation and Test in Europe Conference*, page 1266, Nice, France, Apr. 2009. Keynote speach.

[13] S. Ghiasi, T. W. Keller, and F. L. R. III. Scheduling for heterogeneous processors in server systems. In *Conf. Computing Frontiers*, pages 199–210. ACM, 2005.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, 2007. chapter 2 and chapter 4.

[15] A. A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann Publishers, 2005.

[16] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, pages 64–75. IEEE Computer Society, 2004.

[17] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[18] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. pages 1 –12, jan. 2010.

[19] G. Martin. Overview of the mpsoc design challenge. In *Proceedings of the 43rd Design Automation Conference*, pages 274–279, 2006.

[20] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th Design Automation Conference*, pages 22–27, July 2002.

[21] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest. Low cost task migration initiation in a heterogeneous mp-soc. In *Proceedings of the conference on Design, automation and test in Europe*, pages 252–253, March 2005.

[22] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini. Shapes: : a tiled scalable software hardware architecture platform for embedded systems. In *CODES+ISSS*, pages 167–172. ACM, 2006.

[23] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1209–1229, 2006.

[24] J. Sato, M. Imai, T. Hakata, A. Y. Alomary, and N. Hikichi. An integrated design environment for application specific integrated processor. In *ICCD*, pages 414–417. IEEE Computer Society, 1991.

[25] H. Shen and F. Pétrot. Novel task migration framework on configurable heterogeneous mpsoc platforms. In *ASP-DAC*, pages 733–738. IEEE, 2009.

[26] J. M. Smith. A survey of process migration mechanisms. *ACM SIGOPS Operating Systems Review*, 22(3):28–40, 1988.

[27] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):131–143, 1993.

[28] M. S. Squillante and R. D. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 143–155. ACM, 1991.

[29] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. A scalable synthesis methodology for application-specific processors. *IEEE Trans. VLSI Syst.*, 14(11):1175–1188, 2006.

[30] G. Zimmermann. The mimola design system a computer aided digital processor design method. In *DAC '79: Proceedings of the 16th Conference on Design automation*, pages 53–58, Piscataway, NJ, USA, 1979. IEEE Press.

# Forest Fires: improving a Cache Replacement Algorithm

Filipe Montefusco Scoton
filipe.scoton@usp.br
University of Sao Paulo

Mario Donato Marino
mdm9uw@virginia.edu
University of Virginia

Jorge Mamoru Kobayashi
jmamoru@regulus.pcs.usp.br
University of Sao Paulo

**Abstract**—A particular value or quantity of an object or event varying inversely as a power of some of its attributes is said to follow a Power Law. In other words, Power Laws state that a few percentage of the causes are responsible for a high percentage of the consequences. The most famous examples are the Pareto Principle, where 20% of a nations population is responsible for 80% of that nations wealth; and the Zipf's Law, that says that the frequency of a word on a natural language written text is inversely proportional to its rank in a frequency table. Another interesting phenomena described by a Power Law is a forest fire caused by a lightning, where there will be few lightnings that will cause massive destruction and many that will burn a few trees. This way, the state of a forest, composed by its fauna and flora, will change according to each lightning. We assume that a program in execution is represented by a sequence of cache contexts, and calls and returns represent changes between different contexts. The adaptation of a cache replacement mechanism should take into a particular context of a program in order to improve locality, reducing the number of misses when switching contexts. With this goal, this work models the cache memory of microprocessors as a forest, comparing the lightnings with context switches during the execution of a program. This way, it would be possible to have a mechanism inside the microprocessor in order to better suit the replacement algorithm of the cache memory to the new context of execution, improving the efficiency when selecting the lines to be evicted from the cache and this way improving performance. Without any loss of generality, we employ the forest fire model to the decay functions that control the adaptivity of the Adapted-Discrete-based Entropy Algorithm - ADEA - cache-replacement mechanism [1]. The adaptivity of the ADEA algorithm can balance between the frequency and recency of use of the cache lines. The results show that for most of the benchmarks, the Forest Fire Switching Mechanism stays between the best and the worst result for each of the decay functions implemented with the ADEA algorithm, showing that it is up to 46% better than the worst ADEA configuration and up to 17% worse than the best ADEA configuration. This means that it is possible to reach a more efficient and stable algorithm, that can be better suited for most of the common applications.

**Index Terms**—Locality, Processor, Cache line, Forest Fire, Power Laws, Information Entropy, SimpleScalar.

## 1 Introduction

According to [4], when the probability of measuring a particular value of some quantity varies inversely as a power of that value, that quantity is said to follow a Power Law, stated by equation 1. Famous examples are known as Zipf's Law [5] or the Pareto Principle [4]. Additionally to those examples, Power Laws can be found in physics, biology, earth and planetary sciences, economics and finance, computer science, demography and the social sciences. [4] gives examples on the distribution of the sizes of cities, earthquakes, solar flares, moon craters and wars.

$$p(x) = Cx^{-\alpha}, \text{with } C = e^{c}. \quad (1)$$

Forest fires can also be modeled as a Power Law [4], where there is a higher chance of having many forest fires that will burn a few trees if compared to large forest fires where most of the trees are burnt. The figure 1 illustrates this idea. In this paper, large forest fires mean big program context switches, like calls and returns, that will be reflected in the way cache memory behaves. The forest fire mechanism will signalize when greater data context-switches happen, helping the replacement algorithm to better replace old addresses for new ones.

Each region of a program has an adequate data recency. Based on the forest fire mechanism, we can build a mechanism that can set the replacement algorithm to be better suited to a specific region of the program, improving data recency.

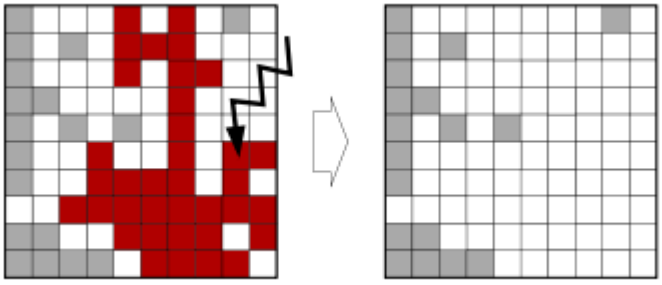


Fig. 1. Forest Fire Model - A lightning strikes a random position in the forest, starting a fire that will cause massive destruction. Figure taken from [4].

Assuming that a program in execution could be represented by a sequence of cache contexts, calls and returns represent changes between different contexts. The cache replacement algorithm should adapt itself on particular contexts in order to improve locality, reducing the number of misses whenever a context switch occurs. The association between lightnings to program calls and returns represents how the idea of Forest Fires modeled as Power Laws can help an adaptive cache replacement mechanism to improve its performance.

We employed the forest fire model to the decay functions

*Mario D. Marino started helping with this work while working as a professor at University of Sao Paulo and now is a doctorate student at University of Virginia.*

used with ADEA - Adapted Discrete-based Entropy Algorithm [1]. The adaptivity given to ADEA by the model can balance between frequency and recency of use of the cache lines. The results show that for most of the benchmarks, the Forest Fire Switching Mechanism stays between the best and the worst result for each of the decay functions implemented with ADEA, showing that it is up to 46% better than the worst ADEA configuration and up to 17% worse than the best ADEA configuration. This means that it is possible to reach a more efficient and stable algorithm, that can be better suited for most of the common applications.

The major contributions of this paper are:

1) Showing how Forest Fires modeled as Power Laws can be associated to an adaptive cache replacement technique in dealing with big context switches;
2) Showing that the technique can be applied with/without the source code, activated on each call/return;
3) Evaluation of the technique on ADEA algorithm, showing its benefits.

Section 2 describes the Adapted-Discrete-based Entropy Algorithm - ADEA - implementation with its decay functions. Then, we have section 3 which describes the Forest Fire Switching Mechanism and implementation. The methodology and results are presented in section 4. Final conclusions will appear at section 5 with some ideas for future work.

## 2 Information Entropy and Controlling Entropy Adaptation

As explained by [1] and stated by [3], for each $x_i$ that belongs to a language, the uncertainty measure can be denoted by the first line of the relation 2. The discrete entropy of a character $h(x_i)$ and the entropy of a character sequence can be further estimated by 2:

$$\begin{aligned} &u(x_i) = -log_b(p(x_i)), b = \{2, e, 10\} \\ &h(x_i) = u(x_i) * p(x_i), i = \{1, \cdots, n\} \\ &H(X = x_i) = \textstyle\sum_{i=1}^{n} p(x_i) * u(x_i) \end{aligned} \tag{2}$$

### 2.1 Adapted-Discrete-based Entropy Algorithm

The Adapted-Discrete-based Entropy Algorithm (ADEA) resides on an independent probability of occurrence of addresses, modeling the working set of a program as a sequence of addresses $X = \{x_1, x_2, \cdots, x_n\}$, where each address $x_i$ is a random variable that can assume any value in the computer address space. The cache replacement policy computes the probability of each address that was referenced during programs execution. This probability $p(x_i)$ is the ratio between the number of times the address was referenced and the total number of references on the cache set that maps the address. With this probability, the algorithm then computes the discrete entropy as $h(x_i)$, described in equations 3, while the original concept of Information Entropy would compute the uncertainty as $u(x_i)$. This adaptation of the Information Entropy original concept is based on a threshold parameter that avoids the inconsistency of the probability of a very frequent access from being 100%, which is shown in 3 below.

$$\begin{aligned} &ts = last\_cache\_set\_access - last\_block\_access(x_i); \\ &decay = 1/(log10(ts+1)+1); \\ &h(x_i) = \begin{cases} u(x_i) * p(x_i) * decay & \text{if } p(x_i) < threshold \\ 1 - (u(x_i) * p(x_i) * decay) & \text{if } p(x_i) >= threshold \end{cases} \\ &\text{For the standard ADEA: } decay = 1; \\ &\text{For RRF-decay, on replacement: } p(x_i) = 0; \text{ if } ts > 10^6 \end{aligned} \tag{3}$$

As explained by [1], ADEA has an ascending slope after the probability $p(x_i)$ reaches higher values than the threshold, which is exactly the opposite behavior than the typical Information Entropy concept. In the case of oddly distributed probabilities among possible values of random variables, $h((x)_i)$ declines from this threshold and above. [1] determines empirically that the best value for the threshold is around 0.38, where the entropy curve presents its inflexion.

Insertion and replacement in ADEA occur in the same position of the cache set stream. The line with the lowest entropy value among lines currently in the cache set will always be stored at the insertion/replacement position. Lines with higher values of entropy will be stored at the other end of the cache set stream.

This way, in ADEA a line is only migrated or promoted to the protected position if it increases its entropy value, which happens if it becomes more referenced; actually this is the greatest dissimilarity between ADEA and LRU algorithms. While LRU immediately inserts the incoming line at the most protected position of the cache set stream, ADEA will wait for more references to the same line to promote it, relying on the probability of future references to that line.

In other words, ADEA presents a higher inertia than LRU when promoting lines to protected regions, because the entropy value increases only with further references. Compared to LRU, ADEA takes longer to decide to remove a line. There may be execution scenarios where any intensively accessed line is moved to the protected position of the cache set and after a while it does not get any further reference. The incoming lines will present a lower entropy than this first line for a period of time.

As described by [1], ADEA may cause a cache line to be indefinitely in the cache, resulting in a waste of space, which is called pinning. This can happen after a high intense access to a particular line compared to other lines and it is the motivation behind the decay functions.

The balance between frequency and recency is controlled by two different decay functions [1]:

- Frequency-recency FR-decay function: relies on frequency and recency to retire cache lines;
- Recency-rather-frequency RRF-decay function: prioritizes recency rather than frequency.

In general, ADEA will assign a discrete entropy value to a cache line that was intensively accessed. In this case, the line would be placed in the most protected area of the cache

and would not be moved from there unless a new cache line shows higher frequency of references. In this case $p(x_i)$ is the regular probability of $x_i$ calculated by the ratio of frequency of references for $x_i$ and number of references for the cache set to which that block is mapped. The variable $u(x_i)$ is the uncertainty as calculated by the equation 3 stated by [1].

FR-decay function [1] employs a combination of frequency and recency of access to calculate the discrete value of entropy. It keeps track of how far in the past the last line reference happened relative to the last cache set access through the time stride parameter, so that an intensively accessed line that is stored in the most protected portion of cache will have its discrete entropy value decreasing with an inversion of logarithm of time 3.

RRF-decay function is intended to cover situations where a highly accessed line was evicted from a cache due to the decay function and later becomes referenced again by the program. If the frequency counters are not reset in the moment where this line is evicted, an extra-offset will be added, pushing the line directly to a protected portion of the cache and leveraging the history of accesses and the newest reference. This may lead to undesirable effects such as competition for slots in the cache and eviction of other lines with fewer, but more recent accesses.

RRF-decay function checks if the replaced line was accessed too far in the past execution and if so, resets its access counters and entropy value. By employing this new heuristic on replacements, it is possible to avoid that highly intensive accessed lines return on the protected portion of the cache, which can be seen on equation 3.

## 3 Forest Fire Switching Mechanism

The context switches are represented by calls and returns during the execution of a program. For every call, there's a possible change of the state of the cache, since patterns of memory accesses tend to change.

As mentioned, the Forest Fire Switching Mechanism models the cache as a forest with each line being a tree. In this model, lightnings mean trees getting burned and disappearing, giving place to other trees, or in the case of caches, the burnt trees mean evicted lines and the lightnings represent the replacement of lines.

It is expected that there will be many replacements due to new addresses accesses, this means that there are lots of lightnings that will burn few trees, or few isolated lines will be replaced many times.

The performance improvement will be achieved by understanding what causes the lightnings that will destroy lots of trees, or in other words, the context switches that will make lots of lines to be evicted in order to give space to new lines. The model explores these changes, helping the replacement algorithm to select which trees, in our case which lines should give room to new ones, lowing the miss rate and improving overall performance.

The forest fire switching mechanism basically switches between RRF-decay and FR-decay, using call and return instructions to do that. This way, the replacement algorithm has its behavior switched to better adapt to those changes. The return of the corresponding call leads to a similar change, since the state tends to change again due to a different pattern of memory accesses. We propose then switching from RRF-decay to FR-decay and vice-versa as the two forest fire main mechanisms.

Two models were created to deal with ADEA decay functions:

1) RRF-FR: for this model, every time there is a call instruction there will be a switch from the RRF-decay function to the FR-decay, getting back to RRF-decay when a return instruction is to be executed;
2) FR-RRF: this behaves in an exactly opposite way from the first one, changing from FR-decay to RRF-decay with a call and again to FR-decay with a return instruction.

## 4 Methodology and Tools

The SimpleScalar simulator was used to model and to evaluate the performance of each algorithm. All the aforementioned structures were inserted into SimpleScalar's cache module to support ADEA's operation as demonstrated by [1]. The switching mechanism was done on top of ADEA's implementation. We simulated a common size of L2 cache while the L1-dcache size remained the same, focusing on the behavior of the higher associative cache in the modeled processor.

As in [1], we have used the OOO-core for its intrinsic higher parallelism. Independently, as the ADEA algorithm, the switching mechanism can also work in caches of in-order cores.

To evaluate the technique, the SPEC CPU2000 benchmark was used with its reference input set. For the matter of comparison, LRU was assumed as the baseline for all the simulations and performance comparisons. Table 1 shows the parameters for the modeled cache.

| Parameter | Value |
|---|---|
| L1 I-cache | 16kB; 32B line size; 4-way LRU; |
| L1 D-cache | 16kB; 32B line size; 4-way LRU; |
| L2 Unified | 1024kB; 64B line size; 8-way (LRU / LIP / BIP / ADEA); |

TABLE 1
Cache Definition

Cache miss rate was used as the criteria to compare the performance of the cache line replacement algorithms. All the three implementations of ADEA were used and LRU is the baseline for comparison purposes, since it is the most common algorithm.

The results shown on figure 2 were obtained simulating 4 billion instructions of 18 SPEC CPU2000 programs, with the three different ADEA default setups as presented by [1] and the two new implementations with the Forest Fire Switching Mechanism. The numbers presented on the graphics cover the miss rate results relative to LRU, which is the baseline for all the performance results. Qureshi's LIP and BIP [2] are also presented for comparison as in [1], but their implementation does not carry the Forest Fire Switching Mechanism, being the same algorithm created

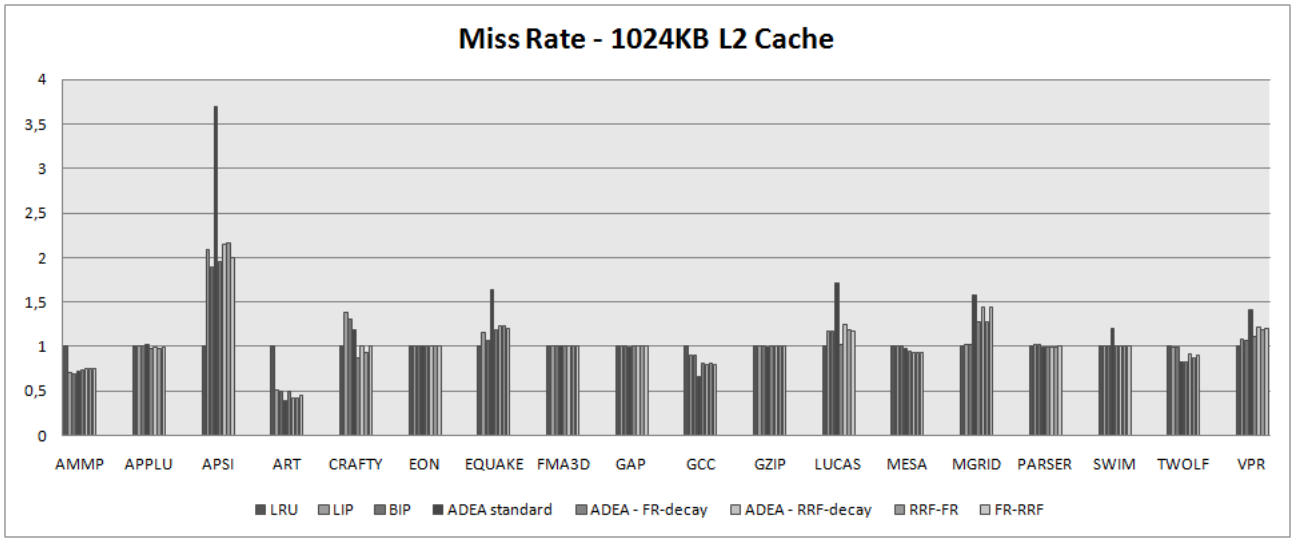



Fig. 2. Miss Rates for an L2 cache with 1024 kB.

by the author and reimplemented on SimpleScalar for this work.

The switching mechanism was able to improve ADEA with no decay function for the benchmarks *applu, apsi, crafty, equake, lucas, mesa, mgrid, swim* and *vpr* while has shown the same performance in the case of *eon, fma3d, gap, gzip* and *parser*. The other benchmarks presented better results with standard ADEA, even compared with FR-decay and RRF-decay, as observed by [1].

As mentioned before, the Forest Fire Switching Mechanism adapts between FR-decay and RRF-decay, going from one algorithm to the other by the trigger of calls and returns. Compared to FR-decay and RRF-decay, the switching mechanism was always equal or between the best performance and the worst among these two. This is an expected result, since both implementations of the Forest Fire are switching between the two decay functions. It is interesting to notice that for some benchmarks, the pairs of similar results change in a way that we can assume that some programs have many calls or spend more instructions inside calls. This can be observed with *crafty, gcc, mesa, mgrid, swim, twolf* and *vpr* that have pairs of results indicating that those programs spend a lot of instructions on calls, for example: the switch from FR-decay to RRF-decay has almost the same results as the RRF-decay model.

The performance improvement that comes with the Forest Fire model helps understand how the context switches affect the behavior of the cache replacement algorithm, since for ADEA frequency and recency are main aspects of each decay function.

## 5 Conclusion, Future Work and Acknowledgements

Analyzing the behavior of SPEC CPU2000 programs when simulating the Forest Fire Switching Mechanism with ADEA and its different decay functions, we could observe that those models can be improved with a change in the way the algorithm behaves whenever a context switch occurs. How to apply the model to other adaptive algorithms is intended for the future. For ADEA we concluded that we should not prioritize recency or frequency, but switch between both depending on the impact of a context switch for a specific program.

The model proposed can help us understand how to identify important characteristics of a program during its execution, without clues given by its source code. The study about how the compiler can help identifying these characteristics and leaving these clues is a subject for a future work. Another idea is to verify the intensity of different kinds of calls, switching between different behaviors according to this intensity, and not at every call as implemented.

In order to improve the analysis, we intend to simulate more instructions and other cache parameters, changing size and associativity of the L2 or even a shared cache in a CMP. With more instructions we want to see not just the overall execution results, but to print statistics along the simulation to better understand how the algorithm changes itself during the switches, or in this case, calls and returns.

The authors would like to thank the feedbacks from the reviewers, professor Jorge Kinoshita from USP for the calls/returns idea and Maria A. G. Marino for the help reviewing the text in english.

## References

[1] "Adapted Discrete-based Entropy Cache Replacement Algorithm" *Submitted to ISPASS 2011*, September 2010.

[2] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 381–391.

[3] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical*, vol. 27, pp. 379–423 and 623–656, 1948.

[4] M. E. J. Newman, "Power laws, Pareto distributions and Zipfs law ," *Contemporary Physics*, vol. 46, pp. 323–351, 2005.

[5] I. Kotera, R. Egawa, H. Takizawa, H.Kobayashi, "Modeling of cache access behavior based on Zipf's law ," *MEDEA '08: Proceedings of the 9th workshop on Memory performance*, pp. 9–15, 2008.

# Integral Parallel Architecture in System-on-Chip Designs

Gheorghe M. Ştefan

*Faculty of Electronics, Tc. and IT, Politehnica University of Bucharest, Bd. Iuliu Maniu, 3-5, Bucharest, Romania*

gstefan@arh.pub.ro

***Abstract* — The ubiquitousness of the parallel computational resources emerges in the rapid growing market of system-on-chip. Both, complex and intense computations are requested for solving the fast expanding functional spectrum of the mobile products. The current approach is unable to provide low area and low power solutions for the increased functional hungry. The proposed Integral Parallel Architecture (IPA) provides >100x increase for *GIPS/Watt* and *GIPS/mm*$^2$ than the current structures. This new approach is based on *ConnexArray*™ technology, developed and tested on real chips, and on the Bubble-free Embedded Architecture for Multithreading (BEAM) execution. It is proposed an IP based model to manage tens of threads and a number of execution or processing units which starts from tens and goes up to thousands.**

## I. Introduction

The SoC domain is driven by two forces:

- the functional spectrum is enlarging, requesting highly complex and high data-intense computation,
- the number of transistors per $cm^2$ of silicon increases, while the possibility to follow this trend is limited by:
- our inability to fill up the size/complexity gap between *making* and *specifying* (the technological developments help us to have more transistors/die, but do not provide us with the techniques to write down more lines of code describing circuits with the corresponding complexity),
- our inability to provide architectural solutions for limiting the energy waste (only structural solutions are provided).

Our solution is based on the following main decisions:

1. To "move" the complexity from the circuit level to the informational level, increasing the weight of embedded computation, substituting as much as possible the ASIC approach with programmable solutions. The functional complexity will come mainly from programming.
2. To segregate the *complex* computation by the *intense* computation [13], in order to optimize independently these two too distinct forms of computation.
3. Because the resulting programmable solution will competes with circuits - "naturally" parallel structures - the engine must be a parallel one.

While for the initial stages of developing embedded computation using sequential architecture was a very good solution, in the actual stage of development parallel computation is a must, and the main problem is: *what kind of parallel architecture is the best fit for embedded computing?* Unfortunately, the answer is*: we need as many kinds as possible, because the diversity of circuits to be emulated* ***efficiently*** *requests a comparable architectural diversity*.

Our proposal takes into account the forms of parallelism which result from the most appropriate computation model to be used as starting point for defining what parallelism means. It is the model of ***partial recursive functions*** proposed by Stephan Kleene [6]. Based on Kleene's model, in [7] and [8] is proposed a new taxonomy for parallel computation. The taxonomy proposed by Michael Flynn [3], and the similar ones, are somehow "artificial", because are based on formal constructs derived from the sequential model of Allan Turing.

## II. Integral Parallel Architecture

In [8] is proved that, according to Kleene's model, the building of a parallel model of computation can be exclusively based on the composition rule having the form:

$$f(x_1, \dots x_m) = g(h_1(x_1, \dots x_m), \dots h_n(x_1, \dots x_m))$$

which is a *n*-sized form (see in Fig. 1 its structural version) which describes two aspects of parallelism: the *synchronic* parallelism of computing *n* functions $h_i$, and the *diachronic* parallelism of pipelining $h_i$ with the reduction function *g*.

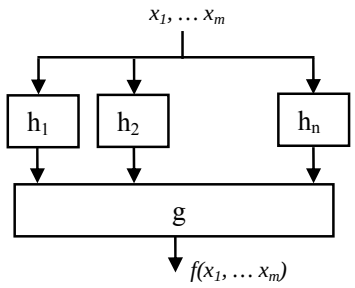



Fig.1. The structural representation for Kleene's composition rule.

Five types of parallel computation can be emphasized:

- **Data-parallel computation**, characterized by:
  $h_i(x_1,\dots x_m) = h(x_i)$, $g(h(x_1),\dots h(x_m)) = \{h(x_1),\dots h(x_m)\}$
- **Time-parallel computation**, characterized by:
  $m = 1$
- **Speculative-parallel computation**, characterized by:
  $h_i(x_1,\dots x_m) = h_i(x)$, $g(h_1(x),\dots h_m(x)) = \{h_1(x),\dots h_m(x)\}$
- **Reduction-parallel computation**, characterized by:
  $h_i(x_1,\dots x_m) = x_i$, $f(x_1,\dots x_m) = g(x_1,\dots x_m)$
- **Thread-parallel computations**, characterized by:
  $h_i(x_1,\dots x_m) = h_i(x_i)$, $g(h_1(x_1),\dots h_m(x_m)) = \{h_1(x_1),\dots h_m(x_m)\}$

Any complex embedded application requests ***all*** these types of parallel computation.

### *A. Implementing IPA*

An IPA is able to perform all types of parallel computation previously listed. The computation in a system with IPA is defined on the following data structures: *scalar*, *vector*, and *stream* of scalars, and uses for defining the computation: *functions* on scalars, vectors or streams (`f(x,…), f(V,…), f(S,…)`) and *function vectors* (`F = <f1 … fm>`).
*We know that*: (1) ***any computation can be expressed using a combination of the following particular forms***:

1. **data-parallel:** $f(V_1 \dots V_n) = V$
2. **reduction-parallel:** $f(V) = x$
3. **speculative-parallel:** $F(x)=<f_1 \dots f_m>(x)=\{f_1(x) \dots f_m(x)\}=V$
4. **time-parallel:** $F(S_I)=<f_1 \dots f_m>([x_1 \dots x_n])=[y_1 \dots y_n]=S$ ; a stream of scalars $[x_1 \dots x_n]$ is applied to the pipe of functions $<f_1 \dots f_m>$; the result stream is $[y_1 \dots y_n]$
5. **thread-parallel:** $f_1(x_1 \dots x_m)=y_1, \dots f_n(x_1 \dots x_p)=y_n$

*We make the assumption that*: (2) ***most of the frequent computations are performed efficiently if they are expressed using a combination of the previously defined functions***.
The assumption (2) is investigated in [9] based on [1]. The sentences (1) and (2) propose a *functional approach* mixed with a sort of *RISC approach* promoted starting with early 1980s. Let's call this approach: ***parallel RISC***.

### *B. Intense computing*

The first four forms of parallel computation have a common characteristic: different kinds of patterns characterize them.

*1. Data-parallel*: each component of the vector results from the predicated execution of the same program.
2. *Reduction-parallel*: each vector component is equivalent related to the reduction function.
3. *Speculative-parallel*: applies, usually, the same variable to slightly different function.
4. *Time-parallel*: a pipe of functions $<f_1 \dots f_m>$ is applied to $[x_1 \dots x_n]$ providing an efficient computation for $n >> m$.

In all these cases the dominant characteristic of computation is its ***intensity***, i.e., a big amount of data is processed or is outputted. Therefore, both, data and program flow are ***highly predictable***, determining the features of the sub-architecture we propose for performing the ***intense computation***:

- the computation is done in a cellular structure of ***many*** *small & simple* processing/execution cells [11]
- array computing is the main type of processing executed in a linear network of cells
- the computation is a high-latency functional pipe
- buffer memory hierarchy with *out-of-core* executions.

### *C. Complex computing*

The multi-threaded computation is a form of parallelism described by: $f_1(x_1 \dots x_m)=y_1, \dots f_n(x_1 \dots x_p)=y_n$, where each function represents a distinct program running on distinct data. Each of these computations is pattern-less. Therefore, we will refer to them as the ***complex computing***, characterized by:

- mono or ***multi*** *big & complex* processor organization
- multi-threaded programming model
- the computation is operating system based
- the memory hierarchy is cache-based.

Faced with intense computation, the current SoCs are designed with few standard complex cores and/or some specific accelerators (DSPs or specialized hardware).

### *D. Integral Parallel Organization*

The first embodiment of a system with an IPA is the ***Connex System*** presented in Fig. 2, where we distinguish between the two kinds of computation, ***segregating*** them as:

- **ConnexArray*™***: many-cell array of execution units (EU) or processing elements (PE) for intense computations [12]
- **Multi-Thread Processor** (MTP) is a *mono-* or *multi*-core **BEAM** processor for complex computations [2].

MTP uses one of its threads to control ***ConnexArray™*** in order to execute an ISA containing instructions for both, scalars and vectors. The entire system is programmed in C++ using the library ***VectorC*** [10]. A GNU C++ compiler is developed for the current IPA instruction set.

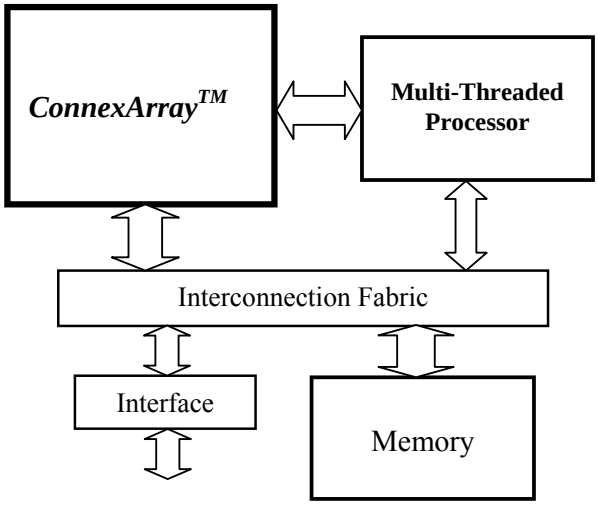



Fig. 2. Integral Parallel Organization: Connex System.

While the intense computation is executed on hundreds or thousands of cores, the complex computation accepts hardly more than 4 cores, because Interconnection Fabric limits less the intense computation. The data stream between Memory and **ConnexArray*™*** is more predictable than the data and program streams flowing between Memory and MTP.

## III. The Complex Computing Part of IPA

The complex part of the computation in Connex System is performed by MTP. Each MTP core is able to execute up to 8 cycle-level interleaved threads. Any active thread is in execution only if its current instruction flow can be executed bubble free. The main effect of BEAM is the increasing of the

effective IPC, while saving the area used for the same purpose in the current processors by the branch predictor, superscalar execution units, and L2 cache. Preliminary evaluations show the increasing of performance by 2.5x – 4x, while the area of the engine is reduced with around 60% [2].

## IV. The Intense Computing Part of IPA

***ConnexArray™*** is a cellular array which performs the intense part of the computation [12], [13]. It is already implemented on silicon in 3 versions. The last one, CA1024 (a SoC for the HDTV market, running at 400 MHz, having 1024 EUs, produced in 65 *nm* standard process in March 2008, see Fig. 4), has the following characteristics measured on actual chips:

- 400 GOPS (**G**iga 16-bit integer **OP**erations per **S**econd)
- **120 GOPS/Watt** and **6.25 GOPS/mm²**

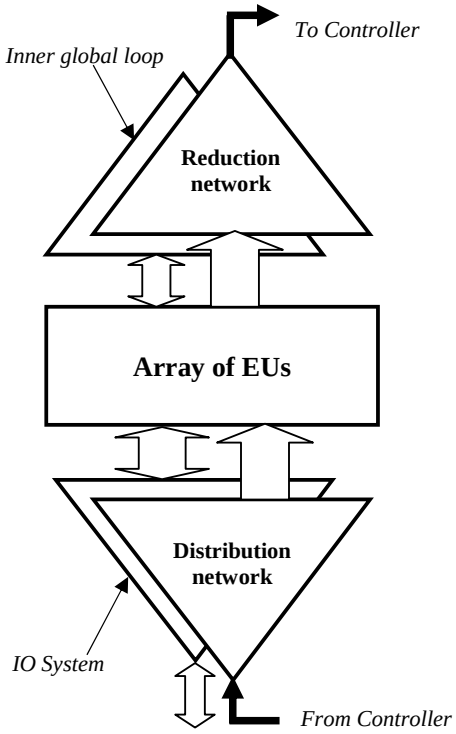



Fig. 3. *ConnexArray™*.

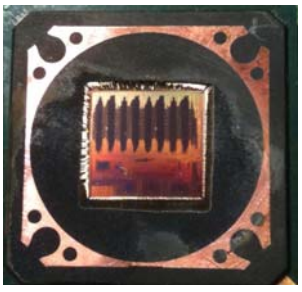


Fig. 4. *CA1024*.

The block diagram of ***ConnexArray™*** is presented in Fig. 3, where a linearly connected array of 1024 EUs receives the same instruction for each EU. The instruction is executed in each EU according with its own state. The reduction network, designed for the most frequently used reduction functions (`add`, `max`, …), sends back to the controller the requested data. An *inner global loop*, closed over the array, is used to classify the EUs according to the selected Boolean. The IO system works in parallel with and transparent to the main computation.

The SoC CA1024 contains besides the 1024 EUs (60% of the chip area) audio/video interfaces, a network of 4 MIPS and a ***time-parallel unit*** (8 16-bit processors).

### A. *Basic Operations in ConnexArray™*

Operations on vectors are performed in constant number of cycles. Generic operations are exemplified in the following:

- **full vector ops**: `{carry, v5} = v4 + v3;` the corresponding integer components of the two operand vectors are added; `carry` is a Boolean vector
- **Boolean operation**: `b7 = b3 & b5;` the two Boolean vectors are ANDed component by component
- **predicated execution**: `v1 = b2 ? v3 - v2 : v1;` in any positions **where** `b2 = 1` the corresponding components are subtracted
- **vector rotate**: `v7 = v7 >> n;` the content of vector `v7` is rotated *n* positions right
- **strided load**: `load v5 addr burst stride;` the content of `v5` is loaded from the address `addr`, using bursts `burst`, on a stride of size `stride`
- **scattered load**: `sld v3 v9 addr stride;` `v3` is loaded indirectly using the address vector `v9`
- **strided store**: `store v7 address burst stride;`
- **gathered store**: `gst v4 v3 addr stride;` it is a sort of *indirect* store.

Each cell contains two sub-cells: the scalar unit and the Boolean unit. For input-output operations there is an IO Plane, distributed over the array, whose content is transferred from or to the array's vector memory in one cycle. On the other hand its content is loaded from or stored to the external memory in a number of cycles depending on the IO latency and bandwidth (around 164 clock cycles for a 400 MHz engine with 1024 16-bit EUs). The transfer process is transparent to the computation.

### B. *VextorC: the programming language of ConnexArray™*

***ConnexArray™*** is programmed in ***VectorC***, a C++ language extension [10]. The extension is made by adding new primitive data types and by extending the existing operators to accept the new data types. In ***VectorC*** the conditional statements have become predication statements.

The new data primitives are, for example:

- **int vector**: vector of integers
- **short vector**: vector of shorts
- **selection**: vector of Booleans

Let be the following variable declarations:

```
int i1, i2, i3;
bool b1, b2, b3;
int vector v1, v2, v3;
selection s1, s2, s3;
```

Then a ***VectorC*** statement like: `v3 = v1 + v2;` replaces:

```
for (int i = 0; i < VECTOR_SIZE; i++)
     v3[i] = v1[i] + v2[i];
```

and `s3 = s1 && s2;` replaces this `for` statement:

```
for (int i = 0; i < VECTOR_SIZE; i++)
     s3[i] = s1[i] && s2[i];
```

The scalar statement: `if (b1) {i3 = i1 + i2};` has the correspondent in ***VectorC*** the vector predication statement:

```
WHERE (s1) {v3 = v1 + v2};
```

replacing this nested `for`:

```
for (int i = 0; i < VECTOR_SIZE; i++)
     if (s1[i]) v3[i] = v1[i] + v2[i];
```

The ***VectorC*** library is used as a programming tool for Connex System and also as a simulation environment.

### C. *Computational performance*

Connex Architecture implements the infrequent, complex instructions, such as multiplication, division, floating point arithmetic instructions using integer resources sequentially. Thus the specific hardware requested for all infrequent operations uses less than 10% from the total area of the array.

This mode of implementing complex operations generates a specific mode of evaluating the performance of the Connex architecture. Claiming the peak performance is meaningless for our architecture, and deceitful for any kind of architecture.

Let's take the example of peak GFLOPS claimed for a typical general purpose processor: 2-4 GFLOPS. There are two factors limiting the peak performance to the effective performance: (1) *the weight of float instructions in current applications* (it is maximum 24% for the most intense float applications, while the medium weight is 18% [4], [5]), (2) *the stalls in the execution pipeline due to the various hazards* (Intel reports from 48% to 85% clock cycles as stall cycles (see http://www.anandtech.com/print/1909)). Results:

$$effectiveGFLOPS = 0.06 \times peakGFLOPS.$$

For Connex architecture the GFLOPS we claim are effective, because the engine uses for float operations exactly as much GOPS as the applications requests. For example, let be a 1024 32-bit cells array running at 1GHz an application which is not IO bounded. Results peak performance of 1 TOPS. The degree of parallelism is in the range of 30% - 90%. Let us take 60%. Then the effective performance is 0.6 TOPS. For a medium float application results the effective performance: 162 GIPS (Giga Instructions Per Second), out of which 29 GFLOPS, and 133 GIPS in integer operations (each floating point operation is executed in 16 clock cycles). Compared with a standard technology, the Connex approach provides more than two magnitude order more effective GFLOPs (from 121x to 243x).

## V. Conclusions

1. The *distinction between complex and intense computation* triggers an efficient segregation which allow two magnitude orders increase for *GOPS/Watt* and *GOPS/mm²* for the intense computation (in ***ConnexArray™***) and one magnitude order for the complex computation (in ***BEAM*** processor).

2. *IPA* expands efficiently the parallel computation at the level of embedded computing by following the golden rule of increasing the size of the design faster than its complexity.

3. Both, intense part and complex part of the system scales with very small performance penalties.

4. The architectural rule of keeping the logic *small & simple*, performing only frequent operations, avoids big, infrequently used active structures.

6. Programmability deserves an increased attention for architects also because the technological costs in nano-era make unmarketable the pure ASIC approach.

## Acknowledgment

The author got a lot of support from main technical contributors to the development of the ***ConnexArray™*** technology, the associated language, and the first applications: Emanuele Altieri, Petronela Bumbăcea, Valeriu Corduneanu, Frank Ho, Radu Hobincu, Mihaela Malița, Bogdan Mîțu, Lucian Petrică, Victor Radu Rădulescu, Marius Stoian, Dominique Thiébaut, Tom Thomson, Dan Tomescu.

## References

[1] K. Asanovic, et al.: *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183.J.

[2] V. Codreanu, R. Hobincu: "Performance Gain from Data and Control Dependency Elimination in Embedded Processors" accepted at *ISETC 2010*. http://phd.arh.pub.ro/resources/beam/isetc2010.pdf

[3] M. Flynn: "Very High-Speed Computing Systems", in *Proceeding of the IEEE*, 54(12), December 1966, p. 1901-1909.

[4] J. Fritts: *Architecture and Compiler Design Issues in Programmable Media Processors*, PhD Thesis, Princeton University, Department of Electrical Engineering Advisor: Prof. Wayne Wolf, 2000.

[5] J. Hennessy, D. Patterson: *Computer Architecture. A Quantitative Approach*, Third edition, Morgan Kaufmann, 2003.

[6] S. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann.*, 1936.

[7] M. Malița, G. Ștefan, D. Thiébaut: "Not Multi- but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *International Workshop on Advanced Low Power Systems* held in conjunction with *21st International Conference on Supercomputing* June 17, 2007 Seattle, WA, USA.

[8] M. Malița, G. Ștefan: "On the Many-Processor Paradigm", in: H. R. Arabina (Ed.): *Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing*, vol. PDPTA'08, 2008.

[9] M. Malița, G. Ștefan: "Integral Parallel Architecture & Berkeley's Motifs", in *ASAP09 - 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 7-9 July, 2009, Boston, MA, USA, pag. 191-194.

[10] B. Mîțu: "C Language Extension for Parallel Processing", BrightScale research report 2008. http://arh.pub.ro/gstefan/VectorC.ppt

[11] G. Ștefan, M. Malița: "Granularity and Complexity in Parallel Systems", in *Proceedings of the 15 IASTED International Conf*, 2004, Marina Del Rey, CA, ISBN 0-88986-391-1, p. 442- 447.

[12] G. Ștefan, A. Sheel, B. Mîțu, T. Thomson, D. Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Stanford University, August , 2006

[13] G. Ștefan: "One-Chip TeraArchitecture", in *Proceedings of the 8th Applications and Principles of Information Science Conference*, Okinawa, Japan on 11-12 January 2009.

[14] D. Thiébaut, M. Malița: "Pipelining the Connex Array," *BARC07*, Boston, Jan., 2007.

# Session II: Computer Architecture - 2

# Confusion by All Means

Muhammad Faisal Iqbal and Lizy K. John
University of Texas at Austin
{iqbal,ljohn}@ece.utexas.edu

***Abstract*—Performance of computers is usually measured by using benchmark suites. There has been a long debate among computer architects on how to aggregate the individual program results to present a summary of performance over the entire suite. Many researchers have criticized the use of Geometric Mean (GM) but SPEC continues to use it to report performance. Mashey [9] has strongly supported the use of GM. According to Mashey, the programs in a benchmark suite like SPEC are samples of some population of programs. It is important that we model the distribution of population correctly before calculating any statistics and making conclusions based on those statistics. Mashey also conjectures that lognormal distribution is a better model than the normal distribution for such benchmark suites. Since GM is the back-transformed average of a lognormal distribution, its use as a measure of central tendency is statistically correct. In this study, we evaluate the correctness of this lognormal assumption using the large repository of performance results for SPEC CPU2006 published on SPEC's website. Utilizing different tests for normality, we find out that although lognormal distribution models the performance results better than the normal distribution, there is a very large percentage of machines which are neither normal nor lognormal. Our study indicates that most of the non-normality is caused by small number of outliers. We study the causes of these outliers and evaluate the use of *Coefficient-of-Variance* to identify outliers. We also present some suggestions on how to deal with these outliers.**

***Index Terms*—Benchmark Means, Geometric Mean, Normality Test, Lognormal Distribution**

## I. Introduction

There is a long history of debate on how to summarize the performance of a benchmark suite and which mean is an appropriate measure of the central tendency [8], [7], [10], [12], [5], [9]. There have been strong arguments both in favor of and against the use of each type of mean. Citron et al. [2] present a detailed history of this discussion. In this section, we discuss some of the arguments made in this regard.

According to Lilja [8] arithmetic mean is proportional to execution time and hence is the right measure for time based metrics. Lilja [8] and Cragon [3] argue that harmonic mean should be used for rate based metrics and weighted arithmetic or harmonic mean should be used for time and rate based metrics respectively if weights of individual programs are different. Both Smith [12] and Lilja [8] strongly oppose the use of geometric mean as a measure of central tendency. They show that although geometric mean produces consistent ordering of machines when normalized times are compared, this ordering is consistently wrong with reference to the total execution times of the benchmarks. Hence they conclude that geometric mean is not the appropriate mean for summarizing times or rates, irrespective of whether they are normalized.

Similarly, John [7] argues that weighted arithmetic or harmonic mean can be used to correctly represent performance. She shows with numerical examples that both arithmetic and harmonic means yield correct orderings with respect to execution times if these means are appropriately weighted. She also maintains that geometric mean is not an appropriate measure since it is not proportional to the execution times of the benchmarks.

On the other hand Fleming et al. [5] study all three types of mean and vote in favor of geometric mean since it always produces consistent rank order among the machines. The most convincing arguments in favor of geometric mean have come from Mashey [9]. He has performed detailed characterization of workload analysis and argues that benchmarks like SPEC's CPU benchmarks are examples of *Sample Estimation of Relative Estimation of Programs (SERPOP)*, i.e., the benchmarks in these suites are samples representing a population of programs which might run on a particular machine. He argues that performance of machines on benchmarks like SPEC can be better modeled using lognormal distribution than the normal distribution. Geometric Mean which is the *back transformed average of lognormal distribution* is the statistically appropriate measure to be used. Also, the analysis done by Lilja and John [8], [7] can be considered as *Workload Analysis With Weights (WAW)* where the user knows exactly which programs will run on the machine and the relative frequency/importance of the programs. In case of WAW analysis, weighted AM or HM are indeed the correct measures for algebraic calculations as pointed out by these researchers. Most of the benchmarking efforts, however, try to do the SERPOP analysis and hence we'll deal with this kind of analysis in the remainder of the paper. We evaluate the correctness of lognormal assumption using SPEC CPU2006 data with three different tests for normality: Lillie Test, Shapiro-Wilks Test and D'Agostino-Pearson Test. We also evaluate the effectiveness of COV in identifying the outliers and present some suggestions on how to deal with these outliers.

## II. Is Geometric Mean an Appropriate Metric?

When comparing performance of machines based on the benchmark results, we are actually comparing distributions of performances. It is important that we understand the nature of these distributions before calculating any statistics and making conclusions based on those statistics. In case of normal distributions, "mean" can be a good measure of central tendency. However, if the distribution is not normal then mean does not give us any useful information about the central tendency and we should be careful while interpreting the

mean. Sometimes a transformation of data can yield a normal distribution and calculation of statistics in the transformed domain can be very useful. In the case of a benchmark suite like SPEC, lognormal distribution is of particular interest. *Lognormal distribution* is the distribution of samples whose logarithm is normally distributed. As Mashey [9] has pointed out, GM can be thought of as the the back-transformed mean of a lognormal distribution

$$GM = \overline{x_g} = (\prod_{i=1}^{n} x_i)^{\frac{1}{n}} = exp(\frac{1}{n}\sum_{i=1}^{n} ln(x_i)) \quad (1)$$

i.e., if we take the mean of logarithm of all samples and then back transform from logarithm, we get the geometric mean. In other words if

$$\overline{x_g} = \frac{1}{n}\sum_{i=1}^{n} log_{10} x_i \quad (2)$$

then

$$Mean = exp(x_g) = GM \quad (3)$$

Thus it is statistically correct to use GM if the data can be modeled using the lognormal distribution. Furthermore, the speedup numbers are calculated as the ratio of execution time of a program on a given machine to execution time on the base machine. There is nothing in the real world that distinguishes base machine A from any other machine B. Ratios of A/B are just as valid as B/A. This is the real fundamental reason why one has to use some metric that works that way, so that if A is 2X faster than B, B should be .5X as fast as A, which only works if we take the logarithms. Arithmetic means of ratios do not have that property, although with small dispersions, normal may be a good quick approximation, and the AM and GM are close anyway. In the case of benchmark suites like SPEC 2006, lognormal distribution can cater for small outliers better than the normal distribution and thus should be a better model for the results. Mashey has shown with one example from SPEC CPU2000 results that lognormal distribution can better model the data. In this paper, we utilize the base results available for SPEC CPU2006 from SPEC's website for about 2000 machines and test how well the normal or lognormal distribution models the data.

### A. Tests for Normality

The easiest and most obvious way of testing for normality is to draw the histogram and visually see how well this histogram resembles the bell-shaped curve. But this is not the most accurate way of testing for normality, especially when the sample sizes are very small as in our case (12 data samples for SPECint and 17 for SPECfp). With small sample sizes, discerning the shape of the histogram is a difficult task and the histogram shape can change significantly just by changing the interval width of the histogram. A better way of testing for normality is to use the normality tests. We perform three different normality tests to verify the assumption of normality for SPEC CPU2006 results. All three tests are *frequentist tests*. Frequentist tests use hypothesis testing and the decision is made using a *null hypothesis*. Null hypothesis is the basic assumption that is put forward when making a statistical inquiry and is usually denoted by $H_0$. The validity of the null hypothesis is tested using the statistical test which calculates a *test-statistic*. In hypothesis testing, the *significance level* $(\alpha)$ is the criterion used for rejecting the null hypothesis. First, the difference between the results of the experiment and the null hypothesis is determined. Then, assuming the null hypothesis is true, the probability (*p-value*) is computed that the difference can be at least as large as observed. If the *p-value* is less than or equal to the significance level($\alpha$), then the null hypothesis is rejected. If the test shows that we should reject the null hypothesis, it is done in favor of an *alternative hypothesis*, represented as $H_1$. In our study the hypothesis testing can be formalized as:

```
H0: Samples are from a Normal Distribution
H1: Samples are not from a Normal Distribution
```

The three tests that we use have different ways of calculating the test statistic and differ in how they quantify the deviation of the actual distribution from a Gaussian distribution. A good discussion on normality tests can be found in [6]. We present a summary of the three tests that we are using:

*1) Lillie Test:* This test is an adaptation of Kolmogrov-Smirnov test with mean and variance of the normal distribution not specified in the null hypothesis. This test first estimates the population mean and variance from the sample data. It then compares the cumulative distribution of samples with the expected cumulative normal distribution. The test statistics is based on the largest discrepancy similar to KS-test, i.e., for a vector x of samples the test statistic is given as

$$KS = max|SCDF(x) - CDF(x)| \quad (4)$$

where SCDF is the empirical cdf estimated from the sample and CDF is the normal cdf with mean and standard deviation equal to sample mean and standard deviation. We performed this test using the `lillietest()` function available in Matlab. We performed a two sided `lillietest()` with an $\alpha$ of 0.05. The result `h` returned by this test is 1 when we can safely reject the null hypothesis, i.e., When the `p-value` calculated by the test is smaller than the significance level $\alpha$.

*2) Shapiro-Wilk Test:* This test is (semi/non) parametric analysis of variance and is useful in detecting broad range of departures from the normality of sampled data. This test is considered to be more powerful in detecting the non-normality than the "distance" tests like the Lillie Test. This test is shown to work for number of samples between 3 and 5000. Most authors agree that this is the most reliable test for normality for small to medium size samples. The test statistic for this test is given as

$$W = \frac{(\sum_{i=1}^{n} a_i x_i)^2}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \quad (5)$$

where $x_i$ are the ordered sample values($x_1$ being the smallest) and $a_i$ are the constants generated from the means, variances and covariances of the ordered statistics of a sample of size

$n$ from a normal distribution. The small values of W are an evidence of departure from normality. This test was also performed in Matlab with $\alpha$ of 0.05.

*3) D'Agostino-Pearson test:* This test assesses the normality using *skewness* (to quantify the asymmetry of the distribution) and *kurtosis* (to quantify the shape, i.e, peakedness of the distribution). A normal distribution is assumed to have a kurtosis value equal to 0. A higher kurtosis means that the distribution is peakier and a negative kurtosis means that the distribution is flatter than the normal distribution. Also, a normal distribution has a skew of zero. A positive skew means that there is a long tail to the right of mean and a negative skew means a tail to the left. D'Agostino-Pearson test first calculates skew and kurtosis of the sample data and then calculates how far each of these values differs from the value expected with a normal distribution. Finally it calculates a single p-value based on these discrepancies. A smaller p-value means departure from the normality. Again, we performed this test using an $\alpha$ of 0.05.

### *B. Do SPEC CPU2006 results follow a Lognormal Distribution?*

We performed all three normality tests for SPEC CPU2006 (both SPECint and SPECfp) results obtained from SPEC's website [1]. The data used in this paper includes all the results which were published on or before September 9, 2010. For normality-testing, we apply the tests on speedup data, i.e., runtime on machine under test/run time on the base machine and for Log normality testing, we use logarithm(speedup) data.

Table I lists the results of normality tests. Columns labeled 'normal' and 'lognormal' represent the number of machines which passed the normality test for speedup and logarithm(speedup) numbers respectively. The numbers given in the two columns are not exclusive, i.e., a machine can be considered both normal and lognormal. The columns labeled "None" show number of machines which were identified as neither normal nor lognormal. We can see that, although lognormal models data better than the normal distribution does, the proportion of machines showing lognormal (or normal) behavior is very small. If the sample values are close to each other, both normal and lognormal assumptions are equally correct to model the data. When the standard deviation increases, i.e., the distributions start having a long tail or a skew, the fit for normal distribution worsens but the lognormal distribution still fits in case of small outliers. Figure 1(a) shows a typical example of a machine whose results (SPECInt in this particular example) show a non-normal behavior. But the logarithm of speedup numbers can be considered normal as identified by Shapiro-Wilk test. Taking logarithm of speedup numbers decreases both skew and kurtosis and brings the distribution closer to an ideal normal distribution. This is typically the case with this category of machines where skew is caused by presence of a small-medium outliers. Figure 1(b) shows an example of the second category of machines. Here the outlier is far away from other programs and even taking logarithm cannot reduce the skew to the desired values.

We also found very small fraction of machines (12/2125) which could be modeled by normal distribution but not by lognormal. In all these cases, taking logarithm made kurtosis negative, resulting in a a distribution which is flatter than a normal distribution. Figure 1(c) shows example of such a machine. Results from D'Agostino-Pearson Test in Table I for SPECint show that percentage of machines exhibiting normal or lognormal behavior is only 13% and 19% respectively. Even with Lillie Test, which is considered the weakest, percentage of normal and lognormal machines is only 16% and 34% respectively. The percentage of lognormal machines is a little higher in case of SPECfp, i.e., 50%, 30% and 25% using Lillie, Shapiro-Wilk and D'Agostino-Pearson Tests respectively. From these results lognormal seems to be a better representation of distribution than normal. In these situations GM is statistically the correct measure of central tendency. But, lognormality cannot be assumed in general as suggested by high percentage of non-lognormal machines in the results. In such situations, the results and statistics should be interpreted very carefully.

### *C. What are the causes of Non-normality?*

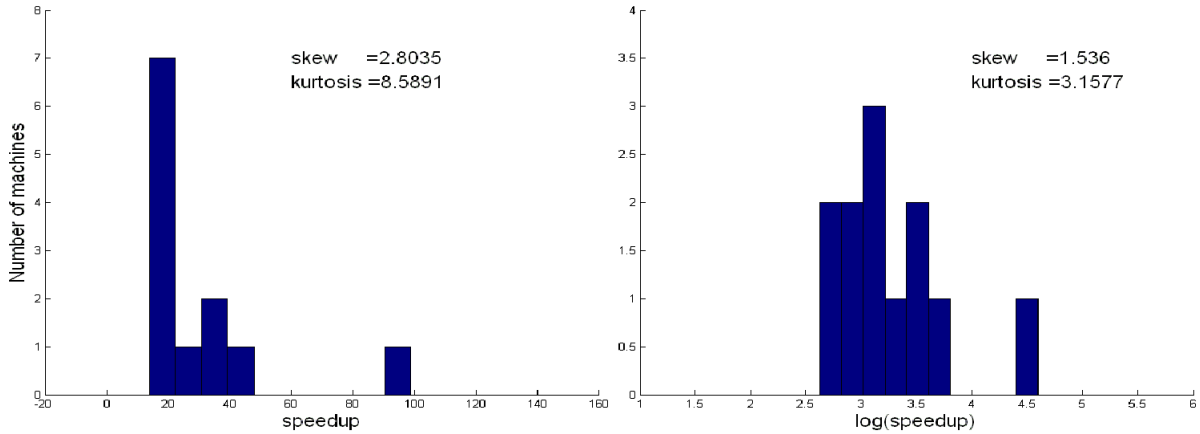
In order to analyze our data set, we calculated the first four moments; Arithmetic Mean, Standard Deviation, Skew and Kurtosis. Then we performed exploratory data analysis to hunt for the odd cases.

*1) SPECint:* Almost all of the machines which show the non-normal behavior have high standard deviation. This high standard deviation is usually caused by presence of an outlier. On a detailed inspection we found that this non-normality is caused by a single outlier, i.e., `462.libquantum`. These machines compile `462.libquantum` with `-parallel` flag enabled. These machines are multi-core machines and support multiple threads, so the performance of libquantum on these machines shoots up. `462.libquantum` is a C library for simulation of quantum mechanics and is easy to parallelize. Part of the speedup also comes from the availability of 64 bit hardware since the benchmark uses 64-bit arithmetic very extensively in its algorithm [4]. In fact, all of the top 10 machines for SPECint have the speedup number for `462.libquantum` greater than $600$. Compiler teams of most of the vendors seem to have cracked this benchmark with compiler flags and cache management instructions. They can focus on just this particular program and get very high values of GM. Thus optimizing for `462.libquantum` is just blowing the numbers away. Such high numbers for one or two outliers badly wreck any statistics approach. Similar things have happened in the past. For example, in the original SPEC89 benchmarks, cache-blocking compilers achieved similar performance gains for `matrix300`. It is important that we identify and isolate the outliers otherwise any statistics calculated with such a data set will not be reliable.

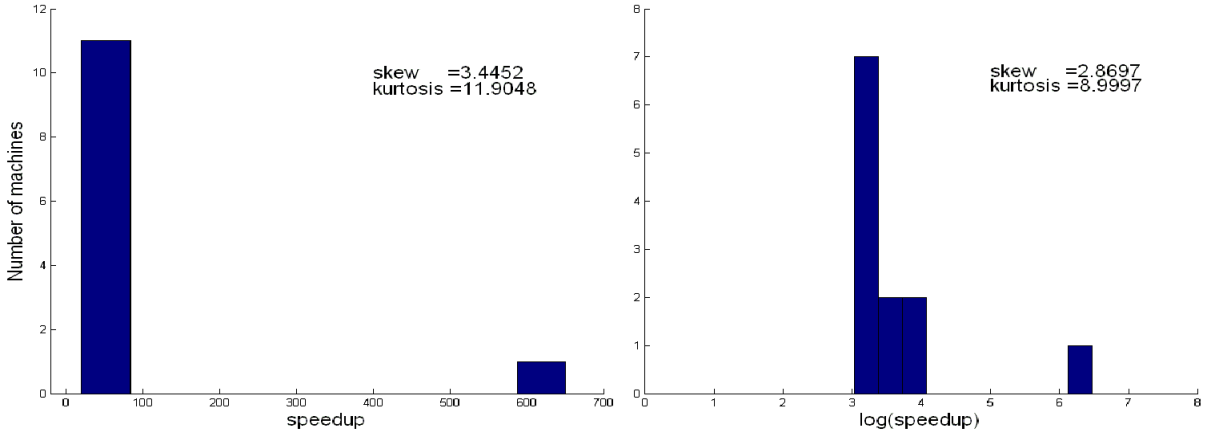
*2) SPECfp:* The situation in SPECfp is not very different. There is a high percentage of machines which show the non-(log)-normal behavior. If we sort the machines with respect to standard deviation, we can easily find out the two outliers in non-normal machines namely `410.bwaves` and `436.CactusADM`. Both these programs are compiled using auto parallelization. The vendors are able to get very high

| | | Lillie Test | | | Shapiro-Wilk Test | | | Dagos-Pearson Test | | |
|---|---|---|---|---|---|---|---|---|---|---|
| benchmark | Total Machines | normal | lognormal | None | normal | lognormal | None | normal | lognormal | None |
| SPECint | 2125 | 341 | 709 | 1415 | 362 | 526 | 1587 | 266 | 398 | 1723 |
| SPECfp | 2066 | 690 | 1469 | 597 | 696 | 1169 | 882 | 565 | 987 | 1057 |

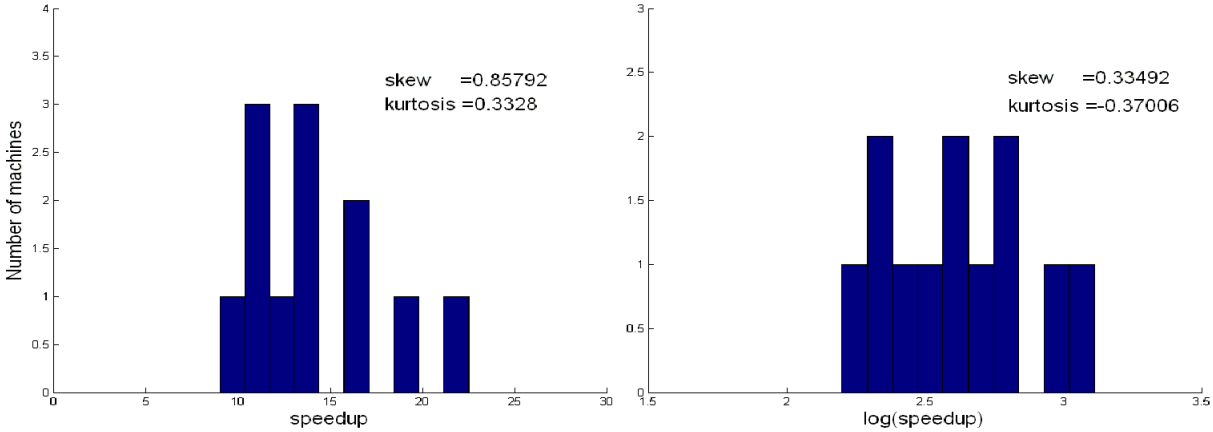
TABLE I
RESULTS OF NORMALITY TESTS FOR SPEC CPU2006




(a) Typical Machine which is non-normal but Lognormal (NovaScaleR410 F2 Intel Core i3-540, 3.06 GHz)




(b) Typical Machine which is neither Normal nor Lognormal (ASUS RS300-E6 (P7F-E) Intel Xeon X3470)




(c) Typical machine which is Normal but not Lognormal (IBM System X 3250 Intel Xeon X3220)

Fig. 1. Histograms of Typical Cases from the SPEC CPU2006 (SPECint) Results

performance numbers for these programs as compared to the other programs. In contrast to SPECint programs which are relatively easy to group, there is a possibility that SPECfp programs need to be categorized into scalar, vectorizable, and parallelizable etc programs. Indeed, programs like `410.bwaves` and `436.cactusADM` do begin to form a second distribution and should be treated separately from other programs. SPEC has encountered similar situations in the past. Initially they began with one single benchmark suite containing both integer and floating point benchmarks. But as soon as they realized the existence of bi-modal distribution in case of integer and floating point programs, they separated the benchmark suite into separate integer (SPECint) and floating point (SPECfp) benchmarks. Similarly SPECfp programs may need to further get split into scalar, vectorizable, parallelizable, and not mixed together. All it takes is one like `410.bwaves` to skew results and badly damage the predictability.

## III. How to deal with Non-normality?

Although lognormal distribution is able to model SPEC2006 data better than the normal distribution, it can do so only in case of small outliers. If the outliers are very far away or there are multi-modal distributions, data cannot be modeled correctly even by lognormal distribution. It is important that we identify these outliers and deal with them accordingly. In this section we present our recommendations to deal with such situations.

### A. Report a Measure of Dispersion

A measure of dispersion should be very useful in identifying the weird cases. It helps in quantifying the ranges and confidence within which to expect most of the benchmarks. Digital Review magazine in 1980s used to report confidence interval, standard deviation and variances for this purpose. In our opinion, Coefficient of Variation (COV) should even be a better measure than standard-deviation and variances. COV is defined as

$$COV = standard\_deviation/mean \tag{6}$$

COV is a better measure because standard deviation must be understood relative to the mean and if one is interested in comparing distributions with different means, co-efficient of variation should be used.

At the moment SPEC gives just one number (GM) and it does not provide any measure of dispersion. Although measure of dispersion can be calculated directly from SPEC's data, a single number like COV can really alert the user about weirdness of results, i.e., outliers or multi-modal distributions. In fact in our results, all the machines which have co-efficient of variance greater than 1 are identified as non-lognormal by all three normality tests. Table II shows the COV of both lognormal and non-lognormal machines in detail. We have used the results of Shapiro-Wilk test for table II.

Fig. 2 plots the COV for SPECint for both lognormal and non-lognormal machines. We can see that COV of all the lognormal machines is less than 1. We found some cases

| | SPECint | | SPECfp | |
|---|---|---|---|---|
| | lognormal | non-lognormal | lognormal | non-lognormal |
| COV(Avg.) | 0.41 | 1.57 | 0.42 | 0.95 |

TABLE II
Average value of COV for Normal and non-Normal machines

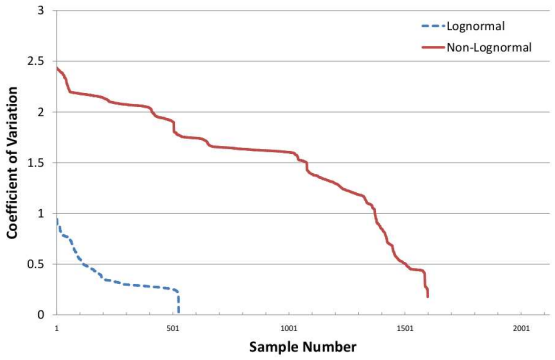



Fig. 2. COV values for Lognormal and non-Lognormal Machines (samples are in decreasing order of COV)

where non-lognormal machines showed small COV. The non-lognormality in these cases is due to high kurtosis (more peakier of distribution) value. This means that more benchmarks are closer to each other. Obviously, if more benchmarks are close to each other, then GM (or any other mean) is a correct measure of central tendency and can be used safely. High COV value always correctly identifies the weird cases of outliers.

### B. Isolate and treat the outliers separately

Once we have identified the outliers, we need to treat them separately from other programs. In case of SPECint, since we find only one outlier, it is easy to just remove it from the stats and use the mean of remaining benchmarks. We removed the outliers, `462.libquantum` from SPECint, `410.bwaves` and `436.cactusADM` from SPECfp and ran the normality tests again. Results are listed in Table III.

From the table we can see that more than 97%, 90% of the machines in both shapiro and Dagos test are lognormal for integer and floating point benchmarks respectively. Thus we can see that after removing the outliers, the distribution can be considered as lognormal and GM can give a good measure of central tendency. The non-normality in the remaining cases is generally due to high kurtosis value. This means that distributions are peakier than the normal distribution and now GM (or any mean) is a good measure of central tendency since we do not have any outliers.

### C. Use a Ranking System not just the Mean

A ranking system can be very useful when a user is comparing multiple alternative machines. Instead of just using

| | | Lillie Test | | Shapiro-Wilk Test | | Dagos-Pearson Test | |
|---|---|---|---|---|---|---|---|
| benchmark | Total Machines | normal | lognormal | normal | lognormal | normal | lognormal |
| SPECint | 2125 | 1137 | 1684 | 2050 | 2112 | 1826 | 2079 |
| SPECfp | 2066 | 1886 | 2017 | 1782 | 1852 | 1683 | 1899 |

TABLE III
RESULTS OF THE NORMALITY TEST AFTER REMOVING THE OUTLIERS

mean to compare the performance of machines, one can use a ranking system like "Borda Counts"[11]. This is a single winner election method and has roots in French Revolution. In this method the voters rank candidates in order of preference. The Borda count selects the winner by giving each candidate a certain number of points corresponding to the position in which he or she is ranked by each voter. The person with most points is declared the winner. In our context, if we are trying to rank `n` alternative machines based on performance of a benchmark suite which has `m` programs, we'll run these `m` programs on all the machines and measure their performances. For every individual program, we compare the performance of each machine and assign points accordingly. Finally sum of points for all `m` programs will decide machine's rank among the `n` alternatives. This type of ranking system is good, since one outlier does not blow away all the statistics. A machine has to perform consistently well in order to be declared as winner. Thus a proper ranking system should be used when we are rank ordering the machine and a single mean should not be used for this purpose.

### D. Generating a Single Number

Computer architects agree that one number like GM or HM can not tell the whole story. But it is sometimes important to get only one number for the purpose of comparisons. We believe that, in these situations, the dispersion of data should be incorporated into this number. One way to do it is to make the final benchmark score inversely proportional to COV. Something like

$$BenchmarkScore \propto (\frac{1}{1+COV})(GM) \tag{7}$$

or

$$BenchmarkScore = (\frac{k}{1+COV})(GM) \tag{8}$$

In this way the machines with high Co-efficient of variance will be penalized more. And the machines in which all the programs perform almost equally will not be penalized. This score number will ensure that nobody will be able to rank better, just by doing program specific optimization on one or two programs of the benchmark suite. More research needs to be done in order to find appropriate values of `k` and to identify more variables that can be incorporated in equation 8.

## IV. CONCLUSIONS

Evaluating multiple machines based on performance of a benchmark suite is generally a SERPOP analysis. In case of small outliers, lognormal is a better model for distribution of performance than normal distribution. The results of normality tests show that lognormal distribution can not be assumed in general. The existence of outliers and multi-modal distributions can badly wreck any statistics approach. With relatively small numbers of benchmarks, it is almost inevitable that there be outliers, and one of the questions raised for future research is: how many benchmarks do you need to improve confidence? A measure of dispersion such as COV can be very useful in identifying such situations. Once an outlier or a multi-modal distribution is identified, one should treat the weird cases very carefully. We also advocate the use of a proper ranking system instead of just the GM in order to rank order the machines. Also, even if a single number is extremely important, the score should take the measure of dispersion into account in addition to the mean as shown in equation 8. A lot of research needs to be done in order to find a proper ranking system and fine tuning the performance score numbers like the one in equation 8.

## ACKNOWLEDGEMENTS

The authors would like to thank John R. Mashey for his valuable feedback. The authors also appreciate the input from Vincent Davis and Youngtaek Kim which helped in improving this manuscript. This work is sponsored in part by National Science Foundation under award 0702694. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of National Science Foundation.

## REFERENCES

[1] Published results for spec cpu 2006. http://www.spec.org/cpu2006/results/.
[2] Daniel Citron, Adham Hurani, and Alaa Gnadrey. The harmonic or geometric mean: does it really matter? *SIGARCH Comput. Archit. News*, 34(4):18–25, 2006.
[3] H. Cragon. *Computer Architecture and Implementation*. Cambridge University Press, 2000.
[4] Dong Ye et al. Performance characterization of spec cpu2006 integer benchmarks on x86-64 architecture. *IISWC*, 2006.
[5] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, 1986.
[6] Zar J. H. *Biostatistical Analysis (2nd edition)*. Prentice-Hall, 1999.
[7] Lizy Kurian John. More on finding a single number to indicate overall performance of a benchmark suite. *ACM Computer Architecture News*, 2004.
[8] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.
[9] John R. Mashey. War of the benchmark means: time for a truce. *SIGARCH Comput. Archit. News*, 32(4):1–14, 2004.
[10] Patterson and Hennessy. *Computer Architecture: The Hardware/Software approach*. Morgan Kaufman Publishers.
[11] Donald G. Sari. Mathematical structure of voting paradoxes, positional voting. *Economic Theory*, 15, 2000.
[12] J. E. Smith. Characterizing computer performance by a single number. *Communications of ACM*, october 1998.

# Validation of Synthetic Benchmarks by Measurement

Jungho Jo, Lizy K. John
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX
jungho.jo@mail.utexas.edu, ljohn@ece.utexas.edu

Michele Reese, Jim Holt
Core Architecture & Modeling Team
Freescale Semiconductor Inc.
Austin, TX
{Michele.Reese, Jim.Holt}@freescale.com

*Abstract* — **Most of the widely used modern benchmarks take weeks to months to finish when executed on cycle accurate simulators, which make it impossible to use them in pre-silicon design evaluations. Processor designers usually rely on the short trace of the workloads or synthetic kernels to determine design tradeoffs. On the other hand, in the customers' point of view, it is impossible to run their realistic applications in the processor design stage. The customers have to wait until the manufacturers tape out their product, which makes it hard to choose their system type.**

**In this paper, we provide an ISA independent framework to generate synthetic benchmarks which replicate the performance characteristics of original programs and validate them on actual hardware by measurement. The synthetic benchmarks are provided in a LLVM compiler's intermediate representative form which can be used to generate binaries of multiple target ISAs. Runtime of the synthetic benchmarks are 10000x times shorter than original while maintaining the performance characteristics so that one can use the synthetic as a proxy for the original benchmarks. The miniaturized clones are validated on a Freescale processor.**

***Keywords-component; Benchmark, Synthetic Benchmark, ISA Independent, Backend Code Generator, Pre-silicon Design Stage***

## I. INTRODUCTION

An ideal set of benchmarks should be representative of modern workloads to reflect the demand of current programs. Many benchmark suites usually consist of modern workloads that are widely-used, representative user programs. For example, SPEC CPU 2006 suite [1] has 29 programs which were carefully chosen from real-life applications. The programs vary in their behavior and language to epitomize concurrent workloads. However, it is very difficult to use these type of benchmarks in pre-silicon design stage evaluations.

The first challenge of using such a benchmark suite in pre-silicon stage is its simulation time. The run time of these programs in a real machine usually varies from few minutes to few hours and is increased many orders when run on a simulator. These days, modern processors have hundreds of millions of transistors in their design. The simulators in the pre-silicon design stage run more than 1000 times slower than real hardware. Previous work shows that it takes months to simulate SPEC CPU2006 workloads on a cycle accurate simulator [2]. It is impractical and almost impossible to use benchmark suites directly with simulators in the pre-silicon design stage.

Another challenge of using the benchmark suite is that how much the suite can be representative of the target application. Even though the programs in benchmark suite are chosen to represent modern workloads, the target application may have its own unique characteristics that are not captured in the benchmark suite. Also, it is beneficial to try as many systems as possible to choose the best platform for the application. Therefore, the best evaluation will be to directly run the target application to assess the performance of the different systems; especially when the application has unique characteristics. However, many software vendors hesitate to do so since most of their application is proprietary software. Especially for software that requires high security, sometimes even giving out the binary is prohibited, since there is risk of the algorithm begin revealed by disassembling the binary.

There have been efforts to address both challenges: simulation time reduction and creating a representative proxy of target application. Bell et al. [3] and Joshi et al. [4] used a technique to create an artificial loop populated with instructions based on a set of profiled metrics of the original program. The synthesized program was used as a proxy for the original program to measure the performance in pre-silicon simulators. These approaches efficiently reduced the number of instructions and also hid the functionality of the original since the instructions were populated in a random manner. The approach can be used to reduce simulation time and also to create proxies for proprietary applications. Prior research [3] [4] showed the efficacy of the approach by generating miniaturized clones of SPEC CPU 2000 suite in the Power Architecture® technology Power ISA and Alpha ISA. Karthik et al. [5] used advanced techniques based on [3] and [4] to create clones of SPEC CPU 2006.

However, results from prior work [3] [4] [5] have a problem in common. Their synthetic benchmarks were generated only for a specific ISA. It is not possible to use the synthetics in other systems that have different ISAs, which results in lack of portability of using the synthetic benchmarks. In early stages of designing a system for a particular purpose, it is important to select the right hardware platform that can satisfy the performance characteristics of the target workload. It is desirable to be able to choose among various platforms by comparing the performance of the target application on each. The problem is that it takes a lot of time and effort to run the benchmarks in multiple targets and it gets even worse when the target processor is still in pre-silicon design stage. It is impossible to run even one benchmark since a single run would

take months. Though pprevious synthetic benchmark approaches efficiently addressed the execution time problem in early design stage, the lack of portability extremely limits their application area to very narrow space, i.e. one specific ISA for which the synthetic code was generated.

Another limitation of previous work is that the synthetic clones of the original benchmark were not validated on actual hardware. All the performance comparison were done using cycle accurate simulators. However, even if they could achieve highly accurate results, it is still questionable whether synthetic clone can be used as a proxy for its original benchmark on a real hardware platform.

The first contribution of this work is to provide a framework to generate miniaturized synthetic benchmarks that can be used in various platforms. As opposed to prior research [3] [4] [5], we generate the synthetic code in an abstract assembly language format provided by LLVM [6]. By using abstract assembly, the synthetic code is not bounded to a specific ISA. Since the abstract assembly is in fact the intermediate representation (IR) of LLVM compiler infrastructure, the synthetic code can be compiled to various target ISAs by using backend compiler of the tool chain. This enables to use our framework to provide synthetic benchmarks in multiple platforms, which enables system designers to choose most suitable hardware for their system even if the hardware is in pre-silicon design stage.

The second contribution is that we validated the efficacy of the synthetic benchmarks on the real hardware platform by measurement. In this work, we used Freescale QorIQ P4080 communications processor that has an e500mc cores. The processor is designed to serve high performance workloads with low power envelope. The e500mc core has four performance counters that can be configured to measure various processor activities. We measured performance characteristics of executions of SPEC CPU 2006 workloads with reference input, having hundreds of billion dynamic number of instructions. Based on the measurement, we provide synthetic clones that have less than 1 million instructions. We compared the performance characteristics of the synthetic benchmarks to the original workload to validate the approach.

## II. Related Work

One of the most commonly used techniques to reduce simulation time is sampling techniques such as [7], [8] and [9]. However, these techniques require either fast-forwarding support from the simulator or huge checkpoint files to reproduce the output. The problem is that it is very inefficient to use fast-forwarding when the interval of execution of interest is located in the later stage of the program execution. Also, checkpoint file requires huge storage space and it is hard to distribute to others. On the other hand, the synthetic benchmark approach provides very small size source code/execution file which is very efficient in run time and storage space.

Another approach to reduce the simulation time is benchmark subsetting [10] which selectively run a subset of benchmark suite whose characteristics are representative of the whole set. This approach is useful when hardware is ready and benchmarking can be finished in a short time. However, in pre-silicon design stage, it is impractical to run even the subset of the benchmark since selected programs are too big to directly run on simulators.

The idea of using statistical simulation to guide the design space exploration was introduced by Oskin et al. [11] and Nussbaum et al.[12]. Eeckhout et al. [13] used the execution frequency of the basic blocks and their transition probability to characterize the control flow behavior of a program. Wong et al. [14] proposed synthesizing benchmark by using the profile of the workload. Joshi et al. [4] proposed creating synthetic benchmarks with microarchitecture independent characteristics. Synthetic benchmarks were generated in embedded assembly format to precisely control the performance.

Low Level Virtual Machine (LLVM) is a compiler infrastructure that supports multiple ISAs [6]. LLVM consists of many modular reusable components that can be built to form a compiler for specific targets. Its core provides source and target independent optimization. It uses code representation known as LLVM intermediate representation (IR) which is human-readable Static Single Assigned (SSA) format based assembly language. LLVM provides various optimization paths that users can easily modify for their purposes.

## III. Synthetic Workload Generation Framework

Synthetic benchmark generation has three major steps. First, we profile the desired metrics from the original workload. Based on the metrics, we generate ISA independent synthetic code in LLVM IR and then generate assembly codes for the target architectures. Finally, we compile the synthetic clone and compare the performance with the original. Fig 1. illustrates the flow of this framework.

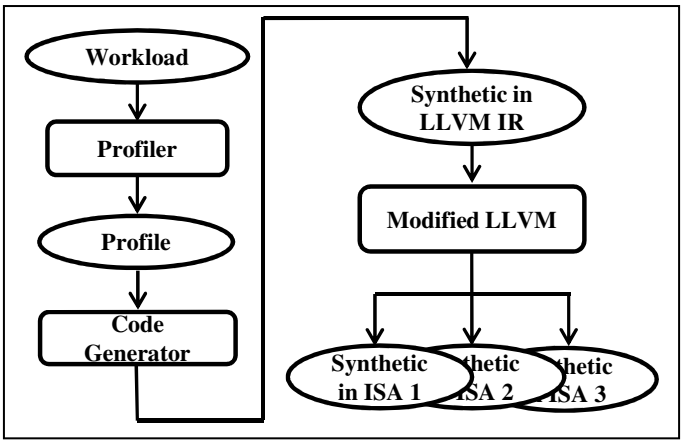



Figure 1. ISA independent synthetic benchmark generation framework.

### A. *Profiling the Metrics*

As the first step to capture the characteristics of the original benchmark programs, we measured properties of the workloads which are shown in Table I. These metrics are categorized into five groups to represent the original program's run-time behavior. We used Freescale's Architecture Description Language (ADL) that models e500mc processor as a profiler. ADL model gives functional execution of the program where

we can attach a plug-in to get the detailed information of each instruction. Some of the microarchitecture dependant characteristics such as branch prediction rate was measured on Freescale's QorIQ P4080 processor by using performance counters. P4080 processor does not provide instruction level granularity since we cannot read performance counters for every instruction.

TABLE I. METRICS PROFILED TO CHARACTERIZE THE WORKLOADS

| # | Metric | Category |
|---|---|---|
| 1 | Dynamic execution frequency of basic blocks | Control flow predictability |
| 2 | Successor informatino of basic blocks | |
| 3 | Transition probabilities in SFG | |
| 4 | Average basic block size | |
| 5 | Branch taken rate for each branch | |
| 6 | Instruction pattern in a basic block | |
| 7 | Branch transition rate for each branch | Branch predictability |
| 8 | % Integer instructions | Instruction mix |
| 9 | % Flating point instructions | |
| 10 | % Load instructions | |
| 11 | % Store instructions | |
| 12 | % Branch instructions | |
| 13 | Dependency distance distribution per type of instructions | Instruction level parallism |
| 14 | Stride value of load and store instructions | Data locality |

The branch transition rate captures how quickly a branch transits between taken and not-taken paths. It indicates how easy or hard a branch predictor can accurately predict the branch. A branch with a low transition-rate usually has higher branch prediction rate since it switches direction less for a given period of time.

Instruction Level Parallelism (ILP) is a metric to determine the extent to which the pipeline is used waiting for data dependency. We capture average register dependency distance distribution for each type of the instruction. Instructions that have immediate operand are considered having zero dependency distance.

Data locality affects the behavior in various levels of memory hierarchy and it has critical impact on performance of the synthetic benchmark. We capture stride values of each load and store instructions and synthesize ten stride values. Data region is modeled as ten arrays in the synthetic; each stride is used in load and store instructions to access corresponding array to capture the characteristics such as cache hit rate in all the levels of cache.

Since running the full-size benchmarks takes huge amount of time even in a functional simulator, we cannot profile whole benchmarks with ADL model. We used Freescale's QorIQ P4080 processor that has an e500mc core. The e500mc core has various performance counters that we can use to characterize the workloads. Though they do not provide detailed profile information at a basic block level granularity, they provide all the required metrics at a whole program granularity. We profiled execution of SPEC CPU 2006's training input set with ADL model to get all the metrics in Table 1 at basic block level granularity. Then, we profiled the execution of SPEC CPU 2006 with reference input set on P4080 processor. We could not capture instruction pattern, branch transition rate, dependency distance and stride value information, since we were not able to read performance counters at such a small granularity.

We used both the results from the ADL profile and the system measurement to generate the synthetics since our goal is to generate synthetics for SPEC CPU 2006 reference input execution. We used all the metrics measured from the P4080 processor and missing metrics are extrapolated from profile from the ADL model. By extrapolating, we sacrifice some accuracy compared to prior work [5] which only used detailed basic block level profiling for a single Simpoint. However, our goal is to clone and to validate the performance of SPEC CPU 2006 with whole reference input run, whereas prior work [5] cloned only a part of the workload.

### *B. Synthetic Code Generation*

After measuring the metrics from the original benchmarks, we parameterize the metrics that are used in the code generator to synthetize clones. The synthetic code generator takes these parameters to create synthetic code by the following algorithm:

1. The number of basic blocks to be generated is calculated based on the instruction footprint (SFG information) of the original workload.

2. The size of each basic block is determined with the help of a random number generated based on a distribution using the average basic block size. Profiled instruction patterns are used to populate instructions in the synthetic basic block. When proper pattern is not found, each instruction in the basic block is randomly generated to match the overall instruction mix.

3. Place branch instructions in the end of the basic blocks to bind them together. We group branches by their transition rate and assign each of them with a register. Place a modulo operation on each register to determine whether a branch is taken or not. One of the branch target of the last basic block points to the first basic block so that the whole synthesized blocks form a single loop.

4. Using the dependency distance distribution for each of the instruction types, each instruction in each basic block is assigned with a producer instruction for each of its operands within the loop. If these producer consumer instructions are not compatible with each other, the algorithm moves up/down one or more instructions until it finds a matching producer for each instruction.

5. Four to eight arrays with size of 40 MB to 80 MB is created to model data segments of the workload. Each of the load/store instructions is configured to have a stride value and assigned to an array. Higher cache miss rate is modeled as a larger stride value to create larger footprint in a given period.

6. Address generating instruction for each load and store instructions are populated. The address of the arrays are incremented by assigned stride value so that the arrays are accessed linearly in execution time.

7. The synthetic code is generated in LLVM IR form which is a Static Single Assignment (SSA) based representation. By using LLVM IR, the synthetic code is not bounded to a specific

ISA, but still be able to represent expressions in a higher level language. The generate code can be compiled in various ISAs by using LLVM's backend compiler.

8. Synthetic code is compiled with modified LLVM's backend compiler to generate assembly code for targeted ISA. Since the synthetic code does not contain any functionally meaningful code, some results from the instructions are not used. These instructions are eliminated in normal LLVM. However, since we need all the instructions to match the performance of the original, we modified the optimization path in LLVM to generate every instruction we synthesized.

### C. *Validation of the Synthetic Clone*

After all the steps are over, the final output is assembly code for the target architecture. We can generate multiple assembly codes if we are to evaluate the synthetic in multiple platforms. We compile the synthetic assembly file with target architecture's general compiler. The synthetic binaries are executed in either simulator of the target system or directly on hardware.

The final synthetic clone is configured to have around 300 thousand dynamic instructions which can be run in a few seconds even in performance model simulators. The results in terms of various performance metrics are compared to that of the original workloads.

## IV. EXPERIMENTS AND RESULTS

We compiled SPEC CPU 2006 suite for Power ISA to run with e500mc core that has a 32 KB Instruction and Data L1 Cache and a private 128 KB L2 Cache with 2GB of DRAM memory. We used gcc-4.5 to compile the binaries and ran them on the Freescale QorIQ P4080 processor with Linux 2.6.1 kernel. We measured performance with performance counter monitoring program to get the performance characteristics. Since the synthetic benchmarks have small number of dynamic instructions, we averaged ten measurement. Since the QorIQ P4080 processor and its infrastructure are optimized to run communication applications, not all of the SPEC CPU 2006 benchmarks can be run on it. We were able to successfully compile and run 19 benchmarks from the suite. Some of results are normalized since the purpose of this work not to show the performance itself but to show comparison between the original and the synthetic workloads.

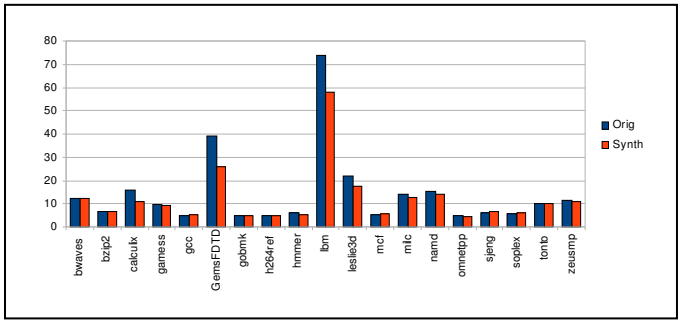



Figure 2. Basic block size comparison of SPEC CPU 2006

Fig. 2 shows the basic block size comparison between the original and the synthetic benchmarks. Some benchmarks with large basic block size have higher error, due to the fact that when the code generation algorithm cannot find compatible dependency for instructions, it reduces the size of the basic block and tries to match dependency. Also, some of the instruction patterns captured by running training input does not appear in the reference input execution or does not have same execution frequency. However, our instruction populating algorithm prioritize training input data, there are some discrepancy in basic block size. Floating point benchmarks have larger basic block size and smaller number of total number of basic blocks, thus they are more sensitive to such errors.

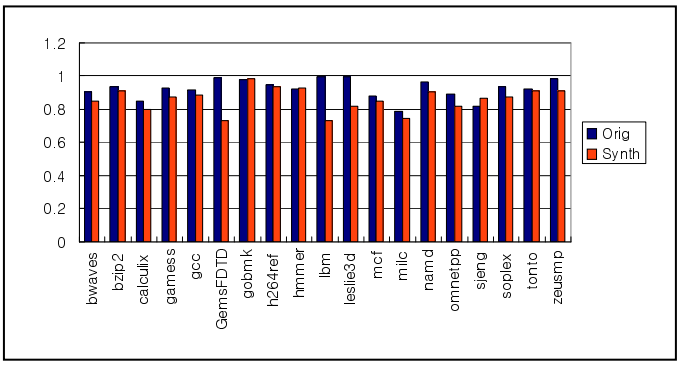



Figure 3. Normalized branch prediction rate comparison of SPEC CPU 2006

In Fig. 3, normalized branch prediction rate is shown. The numbers in the figure are normalized to the highest branch prediction rate of the original benchmark. Average error in branch prediction accuracy is 7.5% with a maximum error of 26.4%. High error occurs in *GemsFDTD, lbm* and *leslie3d* which have small number of integer instructions. The model requires integer instructions to bookkeeping addresses of data access and modular operation of branches. However, when the portion of integer instructions in the original workload is small, the model finds it difficult to generate all the necessary operations. We prioritize the instruction mix in the model, thus we reduce the number of bookkeeping instructions when a workload does not have enough number of integer instructions. It can be noted that benchmarks with higher error belong to the floating point category and reduced bookkeeping information results in high error in branch prediction rate.

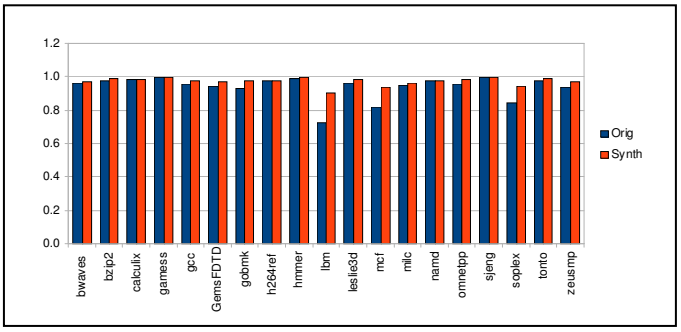



Figure 4. DL1 hit rate comparison of SPEC CPU 2006

Fig 4. shows the DL1 hit rate comparison between the original and the synthetic benchmarks. The average error is 3.8% and the maximum error is 24.1%. The benchmark *lbm* has highest error due to its lack of integer instructions as discussed in the previous section. The model normally maintains ten stride values for load and store to capture the memory access behavior. However, Since when modeling *lbm*, the number of strides was reduced to 4 and memory access patterns were not precisely captured.

The benchmark *mcf* also has high error because the original workload has very large memory footprint. Henning [15] characterized memory footprint of SPEC CPU 2006 and found that *mcf* has stable resident set size (allocated physical memory) of 844 MB and virtual set size (total address space) of 845 MB, which means that *mcf* has very large memory footprint and it accesses all the memory regions throughout its execution. Our framework models the data area as four to eight linear arrays and each of the footprint for the array is 40 to 80 MB. *Mcf* has seven arrays with size of 80 MB where the total footprint size is smaller than the original. It is not only the footprint, but also the memory access pattern that contribute to the error. We are using a linear access model, where the load/store instructions access the addresses in a linear fashion, increasing by their corresponding stride values. This kind of a stride based access behavior causes cache misses in the memory hierarchy, which we use to control the cache miss rate. To achieve high cache miss rate, the stride needs to be very large so that memory accesses result in cache misses. However, the size of array is limited and the stride need to be bounded to prevent overflow, which adds a lower limit to the cache hit rate in the synthetics.

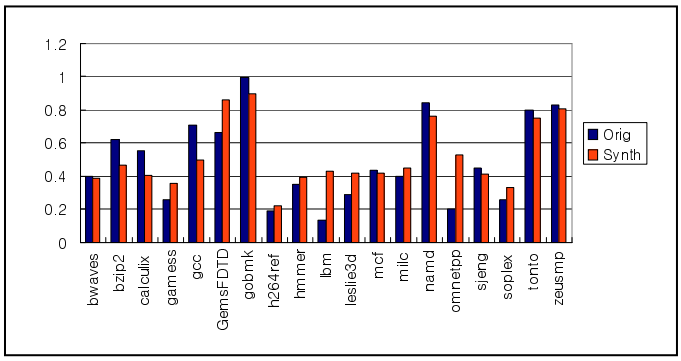



Figure 5. Normalized IPC comparison of SPEC CPU 2006

Fig. 5 shows normalized IPC comparison between the original and the synthetic clones. The numbers are normalized to the highest IPC of the original program. The average error is 37.9% with maximum of 212%. The high errors in IPC mainly occur where the originals have very low IPC around 0.2, which is mainly caused by high DL2 misses and memory load dependencies. Total number of DL2 misses are relatively small and their impact on performance is usually minimal. However, some benchmarks have significantly high DL1 miss rate which causes high DL2 miss rate as well. In that case, access latency to the main memory causes significant IPC drop in the workload, which are not accurately captured in the synthetics. IPC of the synthetic benchmarks are higher than the original for those kind of workloads.

Some benchmarks have pointer-chasing operations that cause very high latency in some benchmarks since load instructions have to wait for another load instructions that causes whole pipeline being stalled. However, we have not yet modeled them because memory contents in the synthetic is not initialized, thus using uninitialized value causes segmentation fault in run time.

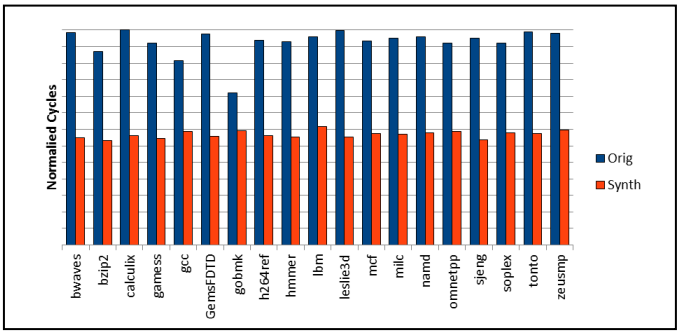



Figure 6. Normalized cycle comparison of SPEC CPU 2006 (Measured cycles shown on Y-axis in logarithmic scale)

Synthetic benchmarks consist of 300 thousand instructions in average and we can achieve significant speed up by using it. Fig. 6. shows normalized cycles to finish the workloads in logarithmic scale. In average, synthetic benchmarks can achieve a speedup of 880,000 in terms of their execution cycles. Such significant speed up well meets the goal of reducing simulation time.

## V. Future Work

Since our model has some limitations to cloning complicated workloads, in this section, we further discuss how to improve the accuracy of the synthetics.

Most of the error comes from the memory access modeling. The model assumes a linearly increasing data address but it does not correctly capture a high miss rate behavior since the stride value is limited to prevent out of bound array access. One way to solve the problem is to make circular memory access pattern. As the synthetic code is forming a loop, we can place instructions to reset the pointer of the array at the end of the loop. By doing so, we can safely use larger stride values to reproduce high cache miss rates.

One of the major goal of this framework is designed to deliver ISA independent synthetic benchmarks. We validated the efficacy of the framework on Power Architecture technology, but this framework can be used in other platforms as well. We are in the process of creating synthetic benchmarks in other ISAs and validating in different platforms.

## VI. Conclusion

In this paper, we proposed a framework that can generate ISA independent synthetic benchmarks. Our framework is able to provide miniaturized synthetic clones to various platforms where running the original workload would take prohibitive execution time.

We evaluated the performance of the generated synthetics by using Freescale QorIQ P4080 processor. The clones of SPEC CPU 2006 suite achieved a speedup of 5 orders of magnitude with an average IPC error of 38%. Further reduction in error is in progress.

## ACKNOWLEDGMENT

This research is an extension of Jo's internship project at Freescale. This work also has been supported and partially funded by SRC under Task ID 1797.001 and NSF under grant number 0702694. Any opinions, findings, conclusions or recommendations expressed in this material are those of authors and do not necessarily reflect the views of the SRC or other sponsors.

## REFERENCES

[1] John L. Henning et al. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News*, Volume 34, No. 4, September 2006.

[2] Karthik Ganesan, Deepak Panwar, and Lizy K John. Generation,validation and analysis of spec cpu2006 simulation points based on branch, memory, and tlb characteristics. *SPEC BenchmarkWorkshop 2009*, Austin, TX, Lecture Notes in Computer Science 5419 Springer pages 121-137, January 2009.

[3] Jr Robert H. Bell, Rajiv R. Bhatia, Lizy K. John, Jeff Stuecheli, John Griswell, Paul Tu, Louis Capps, Anton Blanchard, and Ravel Thai. Automatic Testcase Synthesis and Performance Model Validation for High Performance PowerPC Processors. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2006),* March 2006.

[4] Ajay Joshi, Lieven Eeckhout, Jr. Robert H. Bell, and Lizy K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO),* August 2008.

[5] Karthik Ganesan, Jungho Jo, and Lizy K. John. Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads. *2010 International Symposium on Performance Analysis of Systems and Software (ISPASS 2010),* March 2010.

[6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[7] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program analysis. *Workshop on Modeling, Benchmarking and Simulation,* June 2005.

[8] Greg Hamerly, Erez Perelman, and Brad Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review,* March 2004.

[9] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the International Symposium on Computer Architecture, (ISCA 2003),* p. 84 - 95.

[10] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. *Proceedings of the International Symposium on Computer Architecture (ISCA 2007),* June 2007.

[11] Mark Oskin, Frederic T. Chong, and Matthew Farrens. Hls: Combining statistical and symbolic simulation to guide microprocessor design. *Proceedings of the International Symposium on Computer Architecture (ISCA 2000),* 2000.

[12] Sbastien Nussbaum and James E. Smith. Modeling superscalar processors via statistical simulation. *International Conference on Parallel Architectures and Compilation Techniques (PACT 2001),* 2001.

[13] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. *Proceedings of International Symposium on Computer Architecture, (ISCA 2004),* 2004.

[14] Wing Shing Wong and Robert J. T. Morris. Benchmark synthesis using the lru cache hit function. *IEEE Transactions on Computers,* 1988.

[15] John L. Henning. SPEC CPU2006 memory footprint. *ACM SIGARCH Computer Architecture News,* 2007.

# Selection of Representative Simulation Point using Performance Metric-based Similarity

Satish Raghunath and Byeong Kil Lee

Department of Electrical and Computer Engineering
The University of Texas at San Antonio
*Email: byeong.lee@utsa.edu*

***Abstract*—Design exploration using full simulation of industry standard benchmark (*e.g.*, SPEC CPU 2006 benchmarks) takes long time. The major concern in today's microarchitecture design is reducing simulation time. In this paper, we propose the methodologies to select a single representative simulation point using performance metric-based similarity: (i) instruction mix based simpoint ranking; (ii) metric-based similarity rank using Borda count. We find that the selected single simpoints from the proposed ranking methods show less error rate than the existing single simpoint method.**

## 1. Introduction

Early-stage design exploration requires the detailed simulation, which is executing real world applications on a cycle-level microprocessor simulator. The real world applications are represented by industry standard benchmarks like SPEC CPU 2006 which are called workloads. However, the full simulation of SPEC CPU 2006 benchmarks takes several weeks to months to complete. This problem has motivated several research groups to come up with methodologies to reduce simulation time while maintaining a certain level of accuracy.

Various techniques have been proposed to reduce the simulation time of SPEC CPU 2006 benchmarks [1][4][5]. Among the various techniques to reduce the simulation time, a tool called Simpoint [5] which is based on statistical sampling is popularly used. Simpoint tool employs offline phase classify cation algorithm which calculates the phases for a program/input pair, and then chooses a single representative from each phase and estimates the remaining intervals. The tool chooses this representative for each phase by finding the interval closest to the cluster's centroid using a technique called k-mean clustering. In this paper, we use a standard single simulation point which is extracted from the Simpoint tool that will provide the representative workload as the method of comparison to our method of finding the representative workload. The basic drawback with single simpoint method can be seen in Figure 1, where a standard single simulation point (right-most bar) does not have the lowest error rate to full simulation results. Figure 1 shows the percentage error to the full simulation result with respect to IPC. Some simulation points such as s0, s2, s3, s5, and s11 show better accuracy than standard single simulation point (right-most one). On the other hand, s11 shows the smallest difference (1.7%) while s8 shows the biggest difference (339.3%). Each individual point has its weight information (from the SimPoint tool) which is used for calculating overall metric value with multiple simulation points. In this case, fortunately, s8 has small weight (0.2) which means their impact to overall performance from multiple simulation points is not remarkable. On the other hand, the overall IPC value using multiple simulation points with weight information shows 12.1% error rate to full simulation result. Performance evaluation with standard single simulation point is so closed to the result with multiple simulation points, but it is not the best choice for all metrics.

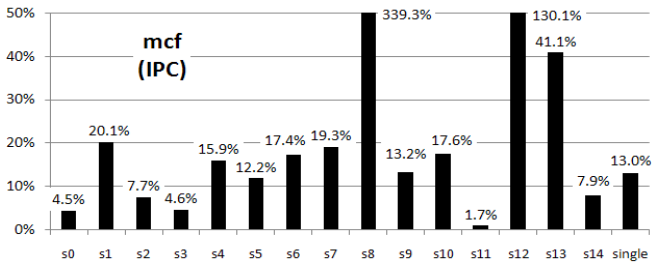



**Figure 1: Error rate of IPC: individual simulation points vs. single simulation point (reference: full simulation)**

In this paper, we propose the statistical methodologies to select a single representative simulation point through the analysis of metric-based similarity and the workload characterization of each individual simulation point. A single simpoint helps in reducing simulation time since it represents the program and leads to faster design cycle. As shown in Figure 1, individual simulation point shows totally different similarity to full simulation result, even though each simulation point is a representative interval having phase information. Two bottom lines of our approach include: (i) architectural behavior of the application is based on probabilistic distribution of instructions as modern computers are instruction-based operation. (ii) Architectural behavior of the application can be expressed as a combination of several performance metrics.

## 2. Related Work

There have been extensive works to reduce the simulation time in microprocessor design [12][13][14][15]. KleinOsowski *et al.* [10] proposed a method to reduce the simulation time of the

SPEC CPU 2000 benchmark suite by using the reduced input data sets. They propose to use small input data sets called *MinneSPEC* that reflect the behavior of the full input data sets instead of using the reference input data sets provided by SPEC. Eeckhout *et al.* [16][17] present their analysis results on the impact of input data sets on program behavior using PCA (principal components analysis) and cluster analysis. Phansalkat *et al.* [8] studied the redundancy of SPEC CPU 2006 benchmark suite based on principal component analysis. Main idea is that SPEC CPU 2006 is biased to some of the applications and simulation time can be reduced by taking benchmarks that are specific to an application. Wunderlichet *et. al.* [14] explains about SMARTS, a trace sampling technique for reducing runtimes in simulators but executions still need to handle tens of millions of instructions. PIN tool from Intel [4][15] is also used for solving the problem of long simulation time.

## 3. Methodology

All the simulations were performed on Intel Xeon processors with a Red Hat Linux operating system. In our experiment, we use both SimPoint [5] and SimpleScalar [21], and performed our simulation with SPEC CPU 2006 Alpha binaries. In order to perform simulation only at the simulation points (which are collected from the Simpoint tool), number of instructions in the program needs to be fastforwarded. Sim-Outorder, which gives a complete report of all the architectural metrics (*e.g.*, cache miss, IPC, power, etc.), is employed throughout this paper for performing the simulations on all the benchmarks [21]. In this paper, we use only 6 benchmarks due to long simulation time needed which is required for full simulation of benchmarks. As a reference to our proposed methods, standard single simpoints are collected from the Simpoint tool with maxK = 1 in K-mean clustering. The number of simpoints generated depends on the value of maxK. For multiple simpoint generation, we use maxK = 30.

## 4. Single Representative Simulation Point

In this paper, two methods have been proposed for finding a single representative simpoint that approximates the full simulation more closely as compared to the traditional standard single simpoint.

### 4.1 I-Mix based single representative simpoint

In general, architectural behaviors depend on the dynamic behaviors of individual instructions in a given workload. Among the instructions, load, store and branch operations tend to have strong impacts to the performance and workload characterization. Figure 2 shows the distribution information of three types of instruction with general–purpose workloads, SPEC CPU 2006 [1]. Considering its significant feature of instruction distribution, it would be a good criterion while choosing appropriate simulation points. We employ the instruction mix information based on the similarity to that of full simulation. I-mix based analysis helps us in identifying the simpoints or phases in a program that have a huge impact from branch instructions.

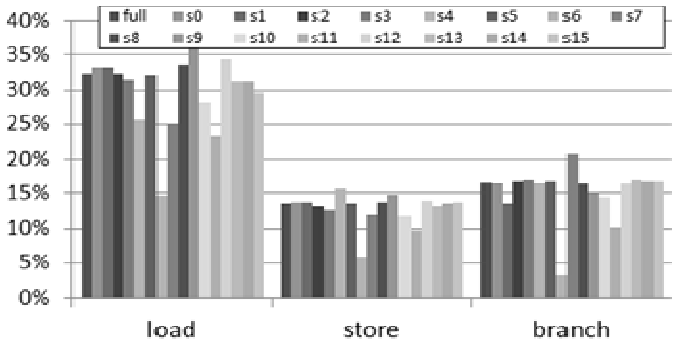



**Figure 2: Comparison of instruction distribution: full simulation vs. individual simulation points**

### 4.2 Metric rank-based single representative simpoint

There are many ways to select a single representative simpoint, but we propose an alternative method to select a metric-aware single simpoint based on ranking method. This method works because workloads can be decomposed into various architectural metrics. Hence, it gives us a better idea of the program behavior and the simpoint generated will have all the metric values close to full simulation. In our research, we are using a concept called as *Borda count* [18][19][20] for ranking the simpoints to choose a representative simpoint. The Borda rule, which is a position based ranking of items, states that the selection of item as a winner is based on "considering many items on an average which item is the highest in ranking" [18]. The ranking is based on the percentage difference of each metric value for each simpoint from the full simulation metric values for each benchmark (*e.g.*, the simpoint with least percentage difference will be ranked the highest priority for that metric). The ranks for each simpoint, considering all the metrics, are then added together and the simpoint with the least rank will be chosen as a representative simpoint. In the Borda method, the item with the highest points or score is the winner. However, in our case, we have modified the Borda count by choosing the simpoint with the lowest score or rank. The above method can be generalized as following equation for a single benchmark.

$$\textbf{Representative Simpoint} = \sum_{i=1}^{k}((simprun(n))k \mid \text{n = 0 to N}$$

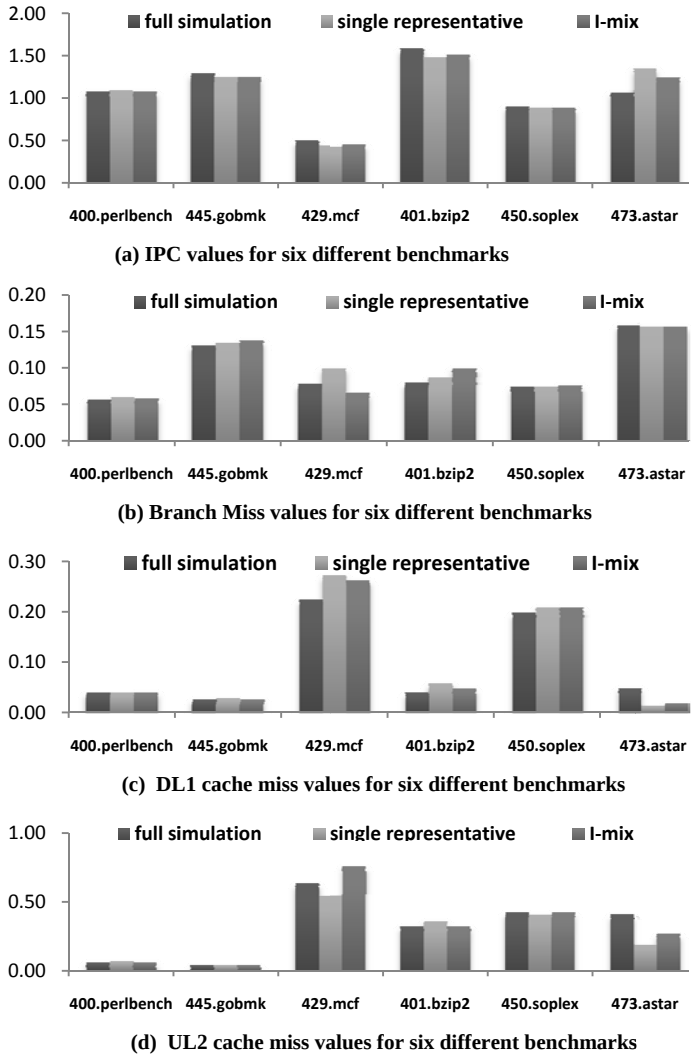
Where k = Number of metrics considered to represent the workload
n = Number of simpoints within each benchmark

## 5. Simulation and Analysis

### 5.1 I-MIX ANALYSIS SIMULATION

In case of I–mix based simulation, we choose the top four most similar I-mix patterns to full simulation. Figure 3 shows the result and the comparison to full simulation and single simpoint simulation. We do not consider il1cache miss rate metric as the metric values are very small and is difficult to represent. We can see from the Figure 3 that the I-mix based simulation shows more accurate result than the single simpoint method. The architectural metrics considered are dl1 cache,

ul2 cache, IPC and Branch miss, but other metrics like average power and leakage power can be considered. Comparing to the single simpoints simulation, IPCs in most of applications show smaller error bound with I-mix based simpoints.




**(a) IPC values for six different benchmarks**

**(b) Branch Miss values for six different benchmarks**

**(c) DL1 cache miss values for six different benchmarks**

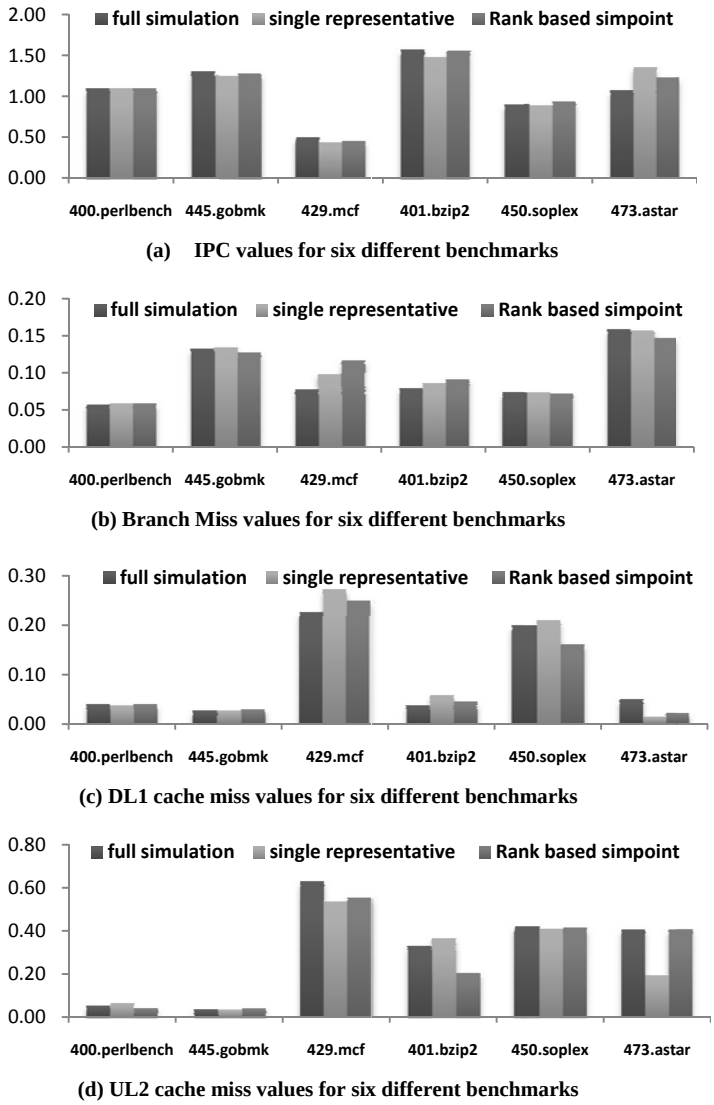
**(d) UL2 cache miss values for six different benchmarks**

**Figure 3: Comparison of I-mix based similarity vs. full simulation and standard single representative**

Figure 3 (a) shows the IPC values for different benchmarks. In the case of 473.astar, the percentage difference between full simulation and I-mix based simpoint method is only 15.94% which is less than single simpoint method and full simulation (26.50%).

### 5.2 Metric-rank based single simpoint Representative

For the rank-based simpoint reordering, the results are indicated from the Figure 4. For most of the benchmarks, all the metrics considered in this method provide a closer value to the full simulation than the single simpoint method. In the Figure 4 (d) which represnts the result for ul2 cache miss rate for the *bzip2* benchmark, more error rate can be seen as compared to single simpoint method. This is because most of the operations in the bzip application take place in the memory. Hence it has more of load and store intructions and therefore it is difficult to characterize a representative phase for such a program. Similar observation is also seen in branch miss for the application bzip2 with I-mix analysis, and mcf with rank based simpoint. In case of 473.astar, the ul2 cache miss rate has a percentage difference of only 0.57% using rank-based simpoint, while the single simpoint method shows 53.76% compared to full simulation result. Some applications like perlbench, soplex and gobmk show a small improvement while mcf and astar show a considerable improvement.




**(a) IPC values for six different benchmarks**

**(b) Branch Miss values for six different benchmarks**

**(c) DL1 cache miss values for six different benchmarks**

**(d) UL2 cache miss values for six different benchmarks**

**Figure 4: Comparison of metric-rank based similarity vs. full simulation and standard single representative**

We now compare the proposed methods to the single simpoint and weighted simpoint method. Table 2 shows the comparison for the metric IPC and gives us the overall quantitative figure for the metric. The overall quantitative figure is obtained from the geometric mean of percentage difference of the metrics values from the full simulation. Table 2 show the geometric mean values for the metric IPC, but it can be extended for other metrics also.

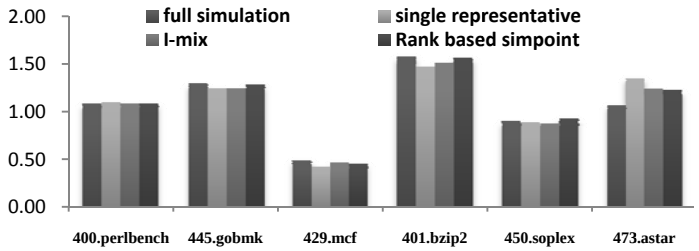



**Figure 5: Comparison between both methods with single simpoint for IPC metric**

We find that the rank based simpoint method has the highest similarity (1.57%) to the full simulation, while the standard single simpoint has the least similarity of 4.52%. Another implication that we can find is that the rank based simpoint outperforms the I-mix based simpoint, but for some metrics the I-mix performs better. From our observation, we see an interesting fact that there exists a subtle difference between i-mix based single representative simpoint and rank based single representative simpoint. On the other hand, workload synthesis is a good alternative in early design exploration. However, when the design is going into lower level, the complexity and difficulties to synthesizing the workload cannot be ignored.

**Table 2. Comparison of both methods for metric IPC**

| Method | Performance |
|---|---|
| Full simulation | *reference* |
| Weighted simpoint (with multiple simpoints) | 2.23% |
| Standard single simpoint | 4.52% |
| I-mix based simpoint | 4.22% |
| Rank based simpoint | 1.57% |

## 6. Conclusion

Full simulation of industry standard benchmarks takes a long time, and sampling methodology is used to reduce the simulation time. In this paper we propose two methods (i) i-mix based and (ii) rank based simpoint for finding the single representative simpoint, and compares them with the traditional single simpoint method. We find that the proposed methods show more similarity to the full simulation result in most of applications and metrics, compared to traditional simpoint method. These methods are now being tested on the other benchmarks in SPEC CPU 2006. We conclude that workload tailoring which are customized for general purpose processors are highly demanded for effective and faster performance evaluation at each design stage.

## References


[1] Standard Performance Evaluation Corporation (SPEC) website, http://www.spec.org/

[2] A. Nair and L. John, "Simulation Points for SPEC 2006," International Conference on Computer Design (ICCD'08). October 2008.

[3] A. Phansalkar, A. Joshi and L. K. John, "Analysis of Redundency and Application Balance in the SPEC CPU2006 Benchmark Suite," The 34th International Symposium on Computer Architecture (ISCA). June 2007.

[4] PIN home page: http://rogue.colorado.edu/Pin/

[5] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Analysis ," Workshop on Modeling, Benchmarking and Simulation, June 2005

[6] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior," Proc. International Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 45–57, Oct. 2002.

[7] http://cseweb.ucsd.edu/~calder/simpoint/single-sim-pionts.htm

[8] A. Phansalkar, A. Joshi and L. K. John, "Analysis of Redundency and Application Balance in the SPEC CPU2006 Benchmark Suite," The 34th International Symposium on Computer Architecture (ISCA). June 2007.

[9] L. Eeckhout, R. H. Bell, B. Stougie, K. Bosschere and L. K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," ISCA. pp. 350-361 2004

[10] A. J. KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," Computer Architecture Letters, vol.1, May, 2002.

[11] K. Lee, S. Evans, and S. Cho "Accurately Approximating Superscalar Processor Performance from Traces," Proceedings of the ISPASS, pp. 238~248, Boston, Massachusetts, April, 2009

[12] K. Ganesan, J. Jo, and L. K. John, "Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads," ISPASS, March, 2010

[13] Timothy Sherwood and Brad Calder, "Time Varying Behavior of Programs," UC San Diego Technical Report UCSD-CS99-630, 1999

[14] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," Proceedings. 30th Annual International Symposium on Computer Architecture, pp. 84-95 ,June, 2003.

[15] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," In Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture, 2004

[16] L. Eeckhout, H. Vandierendonck and K. Bosschere, "Quantifying the Impact of Input Data Sets on Program Behavior and its Applications," Journal of Instruction-Level Parallelism, vol. 5, pp. 1-33, 2003.

[17] L. Eeckhout, R. H. Bell, B. Stougie, K. Bosschere and L. K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," ISCA, pp. 350-361, 2004

[18] I. McLean and N.Shephard,"A program to implement the Condorcet and Borda rules in a small-*n* election," Oxford University. Address: Nuffield College, Oxford OX1 1NF, UK

[19] C.Dwork ,R.Kumar ,M.Naor and D.Sivakumar "Rank Aggregation Methods for the Web," International World Wide Web Conference. pp. 613 – 622, 2001

[20] R.Fagin R.Kumar and D. Sivakumar, "Efficient similarity search and classification via Rank aggregation," International Conference on Management of Data.pp. 301 – 312, 2003

[21] D. C. Burger and Todd M. Austin, "The Simplescalar Tool Set, Version 2.0," UW Madison Computer Sciences Technical Report #1342, 1997.

[22] Timothy Sherwood and Brad Calder, "Time Varying Behavior of Programs," UC San Diego Technical Report UCSD-CS99-630, 1999

[23] D. B. Noonburg and J. P. Shen. "A Framework for Statistical Modeling of Superscalar Processor Performance," Proc. Int'l Symp. High-Performance Computer Architecture (HPCA), pp. 298–309, Feb.1997.

[24] C. Luk, R. zohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05. ACM, pp. 190-200, 2005.

[25] R. H. Bell and L. K. John, "Improved Automatic Testcase Synthesis for Performance Model Validation," 19th ACM International Conference on Supercomputing, June 2005

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T.. Austin, T. Mudge and R. B. Brown, "Mibench: a free, commercially representative embedded benchmark suite," IEEE International Workshop on Workload Characterization, pp. 3-14, Dec. 2001.

# Marching Memory: designing computers to avoid the Memory Bottleneck

Tadao Nakamura
Dept. of Information and Computer Sci.
Keio University
Yokohama, 223-8522 Japan
Email: nakamura@pipelining.jp

Michael J. Flynn
Dept. of Electrical Engineering
Stanford University
Stanford, California 94305 USA

***Abstract*— Marching memory integrates all memory including cache memory and register files into a single unit to avoid the memory bottleneck. Marching memory is organized to synchronize memory columns in minimizing the wire length between memory cells and the operational units as much as possible. A side benefit is lower energy consumption in a smaller packaging format.**

***Keywords-marching memory; memory bottleneck; bandwidth; CPU; DRAM***

## I. INTRODUCTION

This paper introduces a novel memory, the Marching Memory, and the implications for computer organization, Section II discusses the concept of Marching Memory. Section III describes possible Marching Memory circuitry. Section IV discusses the marching memory uses. Section V discusses Complex Marching Memory and Section VI summarizes the future work and challenges ahead to make a practical realization of this technology.

## II. MARCHING MEMORY AND ITS CONCEPT

Marching memory removes the memory bottleneck [1] also called the memory wall [2] by designing a memory so that its access time corresponds to the cycle time of the executing processor. The basic idea is to create a memory structure wherein the data is scheduled to arrive at a fixed physical memory port for immediate use by the processor's functional units. Essentially the data comes to the processor rather than the processor searching randomly for the data.

Fig. 1 shows a basic concept of marching memory with the operation time (column access rate) equal to the CPU's clock cycle. The goal is to have the speed of a L1 cache but provide much larger size. Essentially this memory is designed to support vector and streaming data processing.

The information flow in the marching memory is arranged in streams marching bilaterally (either left or right) across memory columns with one (and only one) column being available to the CPU at any cycle, so that we have the information flow has constant bandwidth (bits/sec) [3] through the organization. The CPU in figure 1 is non specific; it could be an SIMD array processor, a vector processor, a streaming graphics processor, etc. While the interface between the marching memory and the CPU is intended to be as simple and direct as possible; it may be possible to include an interconnect network to route data in the memory column to specific destination functional units as long as cycle time constraints are met.

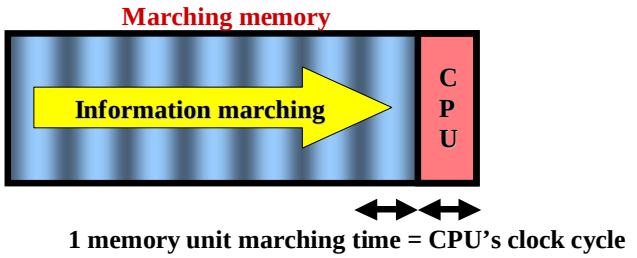



Figure 1. Basic concept of marching memory.

The premise of the marching memory is that the access time to any element in a particular designated memory column is the same as the processor cycle. Fig. 2 contrasts the marching memory and a conventional organization.

Essentially memory access speed decrease as its size grows larger; increasing the memory access latency and decreasing the bandwidth. On the other hand, in marching memory all transfers and accesses are local to adjacent memory columns so there is no change in the available bandwidth as the marching memory size increases.

## III. SCHEME OF MARCHING MEMORY

In its simplest form marching memory can only access adjacent memory columns. So data structures must be carefully scheduled before execution. This is not unlike trace scheduling for instructions in VLIW architectures only here we schedule the data stream. This scheduling requires the application to have a well defined, static data flow graph. In cases where this is not true we need a more general (if slower) form of the marching memory.

This results in two implementations. One is for pure streaming data / vector data for SIMD processing mode, and called simple marching memory. The other one includes a

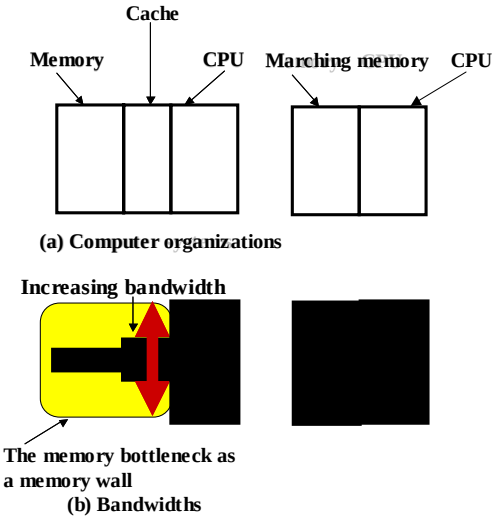



Figure 2. Marching memory and a conventional memory.

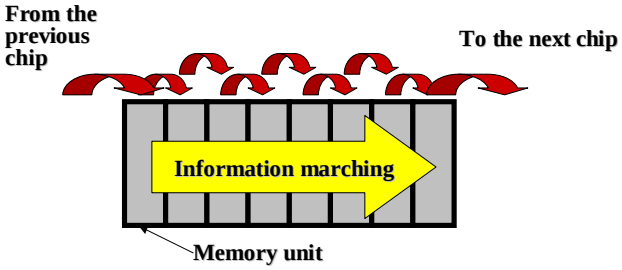



Figure 3. Hardware scheme of marching memory.

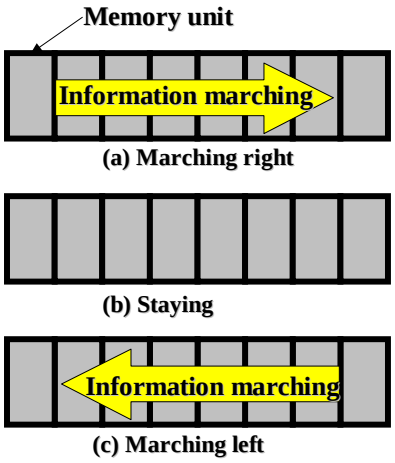



Figure 4. Three modes of marching memory.

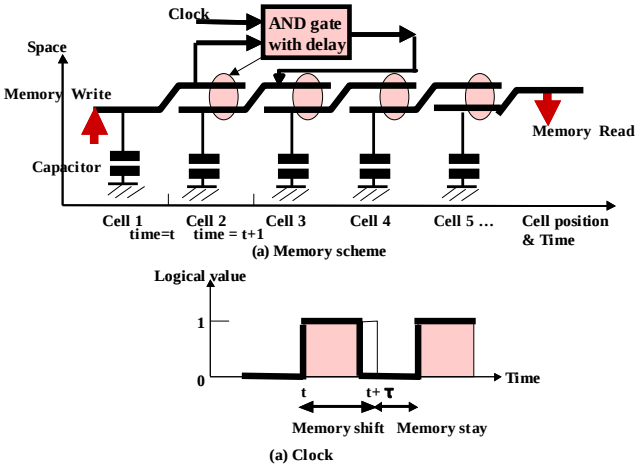



Figure 5. Logic implementation of a one-way marching memory.

mode for random access in either programs or scalar data, and is called complex marching memory.

A more detailed description of simple marching memory is shown in Fig. 3. Information consists of data / instructions is processed by the marching memory. The three modes of behavior are in Fig. 4. Information marching proceeds from left to right or from right to left or the state of staying to process variations in active operation of program instructions and scalar data depending on instructions.

The logic implementing the scheme of a one-direction marching memory of Fig. 3 is drawn in Fig. 5. The circuit timing between stages (in a DRAM type implementation the adjacent column to column transfer time) defines the marching memory stage time. This is assumed to be the same as the element access time within a column. Note that the marching memory is simpler than DRAM [4],[5] because of the absence of long wires for addressing memory units and for accessing data.

As a result, marching memory has a simpler addressing procedure contributing to faster access speed. Power consumption is manageable even though there is significantly more data transfer each cycle. The data is transferred using very short adjacent lines where as DRAM uses long wires with correspondingly large capacitance. The total amount of capacitance switch each cycle remains the same. Moreover, as information is marching, it is usually unnecessary to make refresh the chip (except for long periods in the staying state).

Fig. 6 shows an implementation of marching memory using a switch for the modes and one more set of AND gates with delay marching logic circuitry.

Using the circuit in Fig. 6, we have three modes in Fig. 7 for each of program instructions, scalar data and vector / streaming data processing.

## IV. USES OF MARCHING MEMORY

Vector data and streaming data are easily used in the one directional mode with fewer position indexes corresponding to addresses than in conventional memory. The staying mode corresponds to a memory column access and if the column is buffered at the exit port the buffer acts as a L1 cache.

As mentioned previously marching memory has two types of hardware. One is for pure streaming data / vector data for SIMD processing mode, where the position indexes are used on

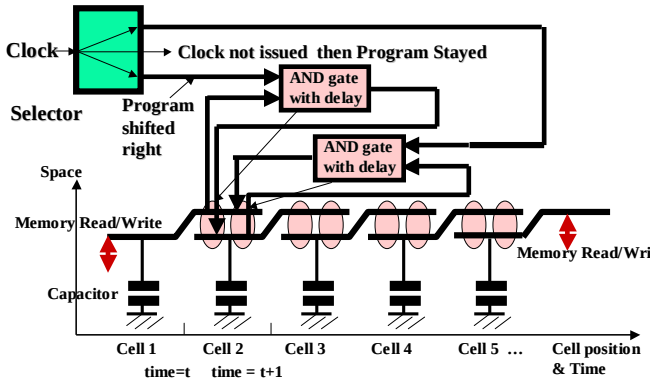



Figure 6. Logic implementation of a bilateral marching memory including a stay mode.

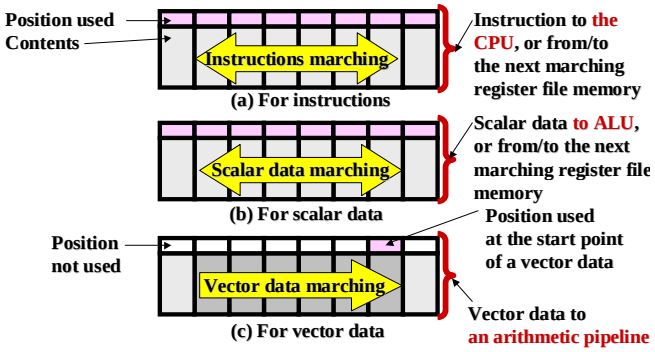



Figure 7. Implementation of the computer organization showing the three modes.

the simple marching memory with the counter in the CPU, which fact is original functionality of marching memory. The other one is for random access mode in either programs or scalar data, where the position indexes are used using address lines on the complex marching memory. In such a case, address wires remain in DRAM's structure.

As a goal, the speed of simple marching memory is equal to that of CPUs machine clock speed. So compared to the speed with that of conventional DRAM memory, the marching memory advantage is about 100:1 as in Fig. 9(a) [6]. Therefore within one access cycle in conventional memory, at most about 100 times the number of operations is possible if all memory units are used in marching memory. Simple marching memory has no long wires because information / data moves synchronously from adjacent memory column to memory column.

The full uses of all available memory units in marching memory within the cycle of conventional memory does not occur EXCEPT in the cases in vector data or streaming data that fully use the units in marching memory. Now consider the situation in Fig. 9 (b). This case is similar to multi-threaded execution. Even though we do not use all the memory units in marching memory, we save the time compared to the conventional memory. The conventional L1 cache that has almost the same speed as CPU's, however, this speed depends on data locality with a small size memory. On the other hand,

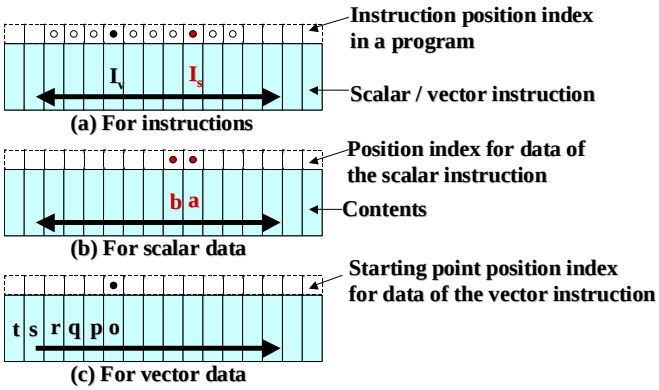



Figure 8. Three configurations for position specifications in marching memory.

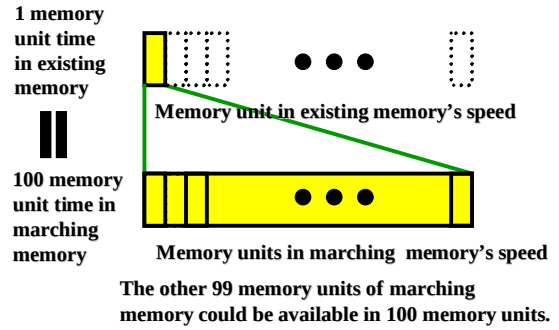
marching memory is useful in case of lower data locality because the bandwidth of the memory is almost constant and the same as CPU's.

## V. Complex Moving Memory

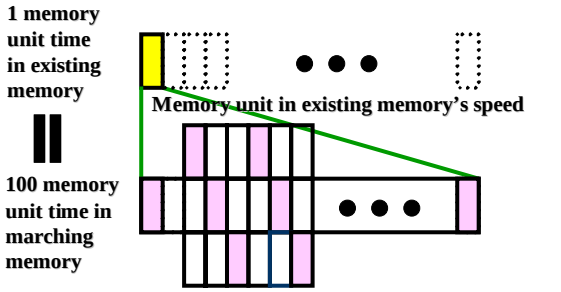
So far we have discussed only the simple marching memory. There are obviously many applications which require a more generalized memory structure as they cannot be perfectly scheduled in a simple marching memory.

The complex marching memory includes a random mode (somewhat akin to DRAM) to enable the addressing of arbitrary columns. This potentially significantly increases the wire lengths as now there is no single fixed physical memory port accessible to the CPU. Indeed the movement (or jump) from one column to another unrelated column is similar to the process to column addressing in a DRAM (the CAS delay). If we now reenter marching mode the delays are longer than in the simple case because the column sense lines are longer. We can partially mitigate this optimizing adjacent column selection and using multiple sub arrays. In the complex type of marching memory it may be better to create hybrid structures which specifically include some simple marching memory arrays.

To address columns in a complex marching memory position indexes are used. These are additional tags to show the location of memory units. For example, at least one position index is necessary for a data item as in Fig. 8(c) if the number of data items is known. However, the memory access is not only regular data structures but also random accesses for program instructions and then scalar data. For these uses, the position indexes are fully activated in preparation of the area added to memory units. Fig. 8(a) shows a configuration of marching memory in storing a program. Here if the program is fully sequential, then the position indexes are not necessary except for the starting one. For branch instructions marching memory has to have position indexes to show the next active instruction in a bilateral marching memory as in Fig. 8(a). The way is also used in scalar data corresponding to conventional instructions. So, the configuration is shown in Fig. 8(b) as well.




(a) The speed gap




(b) A recent general use

Figure 9. Speed gap between marching memory and conventional memory.

## VI. FUTURE WORK AND CHALLENGES

There's a lot of work to be done to make marching memory a viable design alternative, but there's also a significant potential for it.

At the chip level we expect to:

1) Design, simulate and realize a simple marching memory chip; column size less than 1 K bits with 100,000 columns.
2) Design, simulate and realize a complex marching memory of at least the same size as (1). This is the big challenge as the resulting performance parameters will determine the direction of the marching memory project.
3) Carefully study the power management problem.

At the system level we expect to:

1) Create a system simulator for marching memory.
2) Create a compiler to support the scheduling required for the use of the marching memory.
3) Detail the performance of the marching memory in conjunction with various processor architectures and a variety of applications.

The work here has of course next steps to further development for the completion of this chip implementation. First, we are going to make a simulation at a chip level to confirm the behavior in action and secondly at a systematic level to investigate the whole computer organization system including the compiler research to optimize the memory allocation for object codes.

## VII. CONCLUSIONS

We have presented a novel memory with simple access requirements that should be useful in vector and streaming data structure for High Performance Computing and multimedia processing, respectively. For applications that can use marching memory the memory wall before CPUs / arithmetic pipelines is solved. Furthermore, this memory reduces the energy consumption in total computer organizations in compact memory package size owing to the structure of marching memory.

## REFERENCES

[1] M. J. Flynn, "Computer Architecture: Pipelined and parallel processor Design, John & Bartlett Publications, 1995.

[2] W. A. Wulf and Sally A McKee, "Hitting the Memory wall: Implications of the Obvious. Computer Architecture News, 23(1), pp. 20-24, March 1995.

[3] D. Burger, J.R. Goodman and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," Computer Architecture, 1996 Annual International Symposium on pp. 78-78, May 1996.

[4] H. Zheng and Z. Zhu, "Power and Performance Trade-Offs in Contemporary DRAM System Designs for Multicore Processors," IEEE Trans. on COM., vol. 59, No. 8, pp. 1033-1046, 2010.

[5] E Cooper-Balis & Bruce Jacob, "Fine-Grained Activation for Power Reduction in DRAM," IEEE Micro, pp. 34-47, May/Jun 2010.

[6] D. Patterson, T. Anderson, et al, "A Case for Intelligent RAM," IEEE Micro vol. 17, no. 2,pp.34-44, Mar.1997..

# Session III: VLSI Design

# Lightweight Energy Prediction Filters for Solar-Powered Wireless Sensor Networks

*Cory E. Merkel and Dhireesha Kudithipudi and Andres Kwasinski*
Department of Computer Engineering
Rochester Institute of Technology
Rochester, New York 14623-5603
Email: {cem1103, dxkeec, axkeec}@rit.edu

***Abstract*—This research studies lightweight energy prediction filters for solar-powered wireless sensor networks. A generalized prediction filter is developed from the empirical analysis of several solar intensity datasets. The Array of Beta Coefficients (ABC) energy prediction filter is proposed. A comparison metric is also proposed to evaluate different filters based on their accuracy, storage requirements, and calculation complexity. Simulation results show that the ABC filter has up to 8-fold accuracy improvement over other published filters.**

## I. Introduction

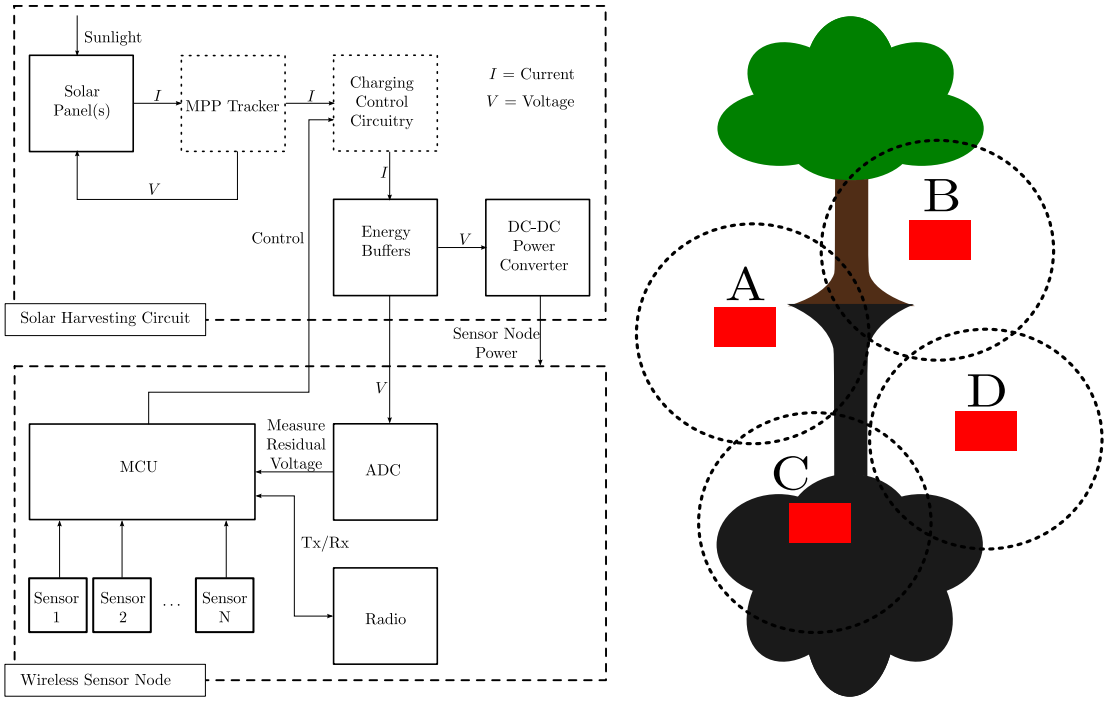
Wireless sensor networks (WSNs) have been proposed for several applications where sensor nodes must be deployed in remote or hostile environments. Some examples are volcanic activity tracking [1], remote habitat monitoring [2], and surveillance of battlefield conditions [3]. In each of these applications, sensor nodes have low physical accessibility. Furthermore, sensor nodes that rely solely on battery power severely limit the longevity of their aggregate network. Therefore, sensor nodes for these applications must be able to harvest energy from an environmental source such as the sun. Figure 1(a) generalizes a wireless sensor node with solar energy harvesting capabilities [4–6]. The solar harvesting circuit converts sunlight to usable energy for the sensor node. Solar panels perform the energy transduction and energy buffers store the converted energy. Other components may be used to track the solar panels' maximum power point (MPP), control the charging/discharging of energy buffers, and convert the stored energy to a usable voltage. The sensor node circuit contains various sensors, a microcontroller, a radio for wireless communication, and interface circuitry such as analog to digital converters.

The sun is a spatially and temporally dynamic energy source. To illustrate this, consider the WSN in Figure 1(b). At the time shown, node C is able to harvest less energy than the other nodes because of the shadow from the tree. Later in the day, however, the shadow may be cast over one of the other sensor nodes. In general, the solar energy available for each sensor node will be a random function of time and space. Therefore, sensor nodes such as those in Figure 1(a) must always be able to estimate the energy that they will have available for a given task. This estimate must be an input to the scheduling, routing, and other WSN algorithms. Otherwise, sensors could be over-utilized (causing them to completely drain their stored energy) or under-utilized (reducing network throughput). Previous works have developed energy estimation algorithms. In [7] the authors developed the Environmental Energy Harvesting Framework (EEHF). In EEHF, an energy estimation is based on two factors. The first is a measurement of the energy already stored in the node. The second is a prediction of how much energy the node can harvest in some future time frame. Again, consider Figure 1(b). If, at the time shown, each sensor node has the same amount of stored energy, then routing data from node A to node D through paths A-B-D or A-C-D has equal overall effect on the total network energy. However, if the path A-C-D is chosen, then the energy lost at C cannot be regained until the shadow moves. If B and C could both predict how much energy they can harvest in a future time frame, then the network will be able to determine that A-B-D is a better route. The prediction method in [7] is a simple autoregressive filter, which exponentially reduces the weight of past energy statistics. In [8], the authors develop the Enhanced-Environmental Energy Harvesting Framework (E-EEHF). E-EEHF improves upon EEHF in several ways, including a more accurate prediction filter.

This work focuses on lightweight prediction filters for solar-powered WSN nodes. We emphasize *lightweight* because complex prediction methods, such as those in the frequency domain or those based on adaptive filtering, are not well-suited for extremely energy-constrained WSNs. To the best of our knowledge, this is the first work that provides a thorough analysis and comparison of lightweight solar energy prediction filters. We also present a new filter called the Array of Beta Coefficients (ABC) filter. The ABC filter is based on the prediction filter used in the E-EEHF framework, but has better accuracy, smaller storage requirements, and less computational demand.

## II. Solar Intensity Analysis and Prediction

In this section we will give a detailed empirical analysis of solar intensity data. A generalized prediction filter is presented for the prediction of solar intensity. Since solar intensity is directly related to solar energy, the generalized prediction filter and all of the filters presented in the following sections can be readily applied to energy prediction rather than solar intensity prediction. The value of solar intensity, $I(t, l)$, at a given time $t$ and location $l$ can be expressed as the sum of its periodic




(a) (b)

Fig. 1. Solar-powered wireless sensor node and example use scenario. In (a), a solar harvesting circuit converts sunlight to voltage for powering the wireless sensor node. In (b) several solar-powered sensor nodes form a network. If node A wishes to transmit to node D, then it must choose the best route based on how much energy will be available to nodes B and C in a future time frame.

and random components:

$$I(t,l) = Per\{I(t,l)\} + Ran\{I(t,l)\} \tag{1}$$

Periodic components result from highly predictable events such as solar and lunar cycles, the movement of shadows from stationary objects (e.g. trees), and the changing of seasons. Random components result from highly unpredictable events such as the movement of shadows from non-stationary objects, cloud movement, and abrupt changes in weather. Solar intensity variation with location may be useful for network algorithms. Our work, however, focuses on individual sensor nodes and, therefore, does not consider the location parameter. Discretizing (1) and ignoring the location parameter $l$ yields

$$I[i] = Per\{I[i]\} + Ran\{I[i]\}. \tag{2}$$

In (2), $i$ is a discrete timeslot representing a time interval $\triangle t$ and is defined as $i = \lfloor \frac{t}{\triangle t} \rfloor$. In this work, we will consider $I[i]$ to be the average solar intensity for the $i^{th}$ timeslot. Solar intensity depends on the location of the sun. Therefore, it is periodic with period $T = 24$ hours. We also define $N_R = \frac{T}{\triangle t}$ as the number of discrete timeslots within $T$. The $R$ subscript in $N_R$ stands for *rounds*, a term borrowed from [8]. Figure 2 illustrates the relationship between the defined parameters. The $i+1$ timeslot's intensity can be predicted from its periodic and random components, which can be defined as functions of intensities from previous periods and recent timeslots. Specifically,

$$Per\{I[i]\} = p(I,i), \tag{3}$$

$$Ran\{I[t]\} = r(I,i), \tag{4}$$

and

$$I_p[i+1] = f(p,r), \tag{5}$$

where $p$ and $r$ are functions that operate on the past data, $f$ is a function that combines the periodic and random data, and $I_p[i+1]$ is the predicted intensity in the next timeslot. Now, we can completely specify an arbitrary filter by defining $f$, $p$, $r$, $N_R$, and $T$. The functions $f$, $p$, and $r$ are chosen by the filter designer based on different design constraints. The period of the data $T$ will typically be 24 hours for solar applications. In the next section, we will discuss how to choose the parameter $N_R$ based on the consideration of several tradeoffs.

## III. Choosing a Timeslot Size

Choosing an optimal time interval or timeslot size $\triangle t$ is a multivariable problem. Figure 3(a) shows some costs as functions of the timeslot size. Calculating an optimal $\triangle t$ in real-time could be very costly. Therefore, we propose a pre-deployment empirical solution based on the usefulness (how much it aids our prediction) of periodic and random data. Due to time correlation in the light intensity, recent data is more useful when the size of the timeslot is small. Conversely, data from past periods is generally more useful when the size of

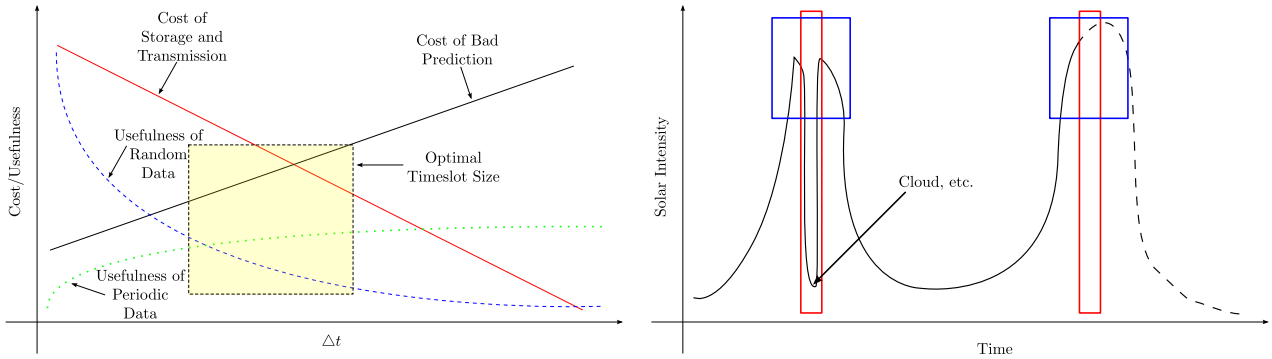



Fig. 3. The choice of $\triangle t$ as a multivariable optimization problem. Several costs as functions of timeslot size are given in (a), and (b) gives an example of how very small timeslot sizes are more sensitive to variation among periods.

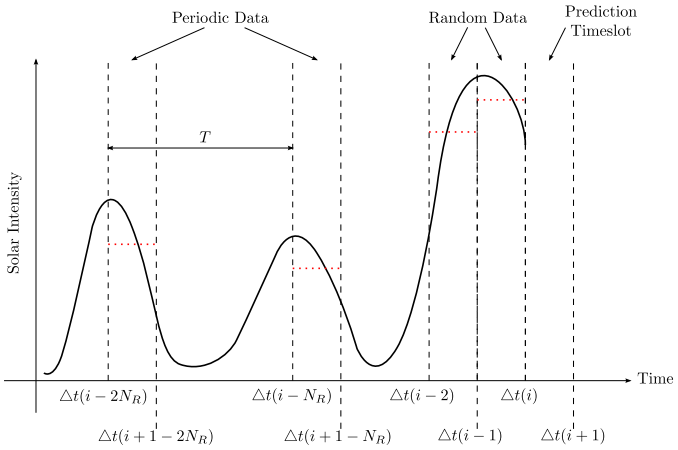



Fig. 2. Solar intensity characteristics. Solar intensity statistics from past periods (periodic data) and recent timeslots (random data) can be used to predict solar intensity in a future timeslot.

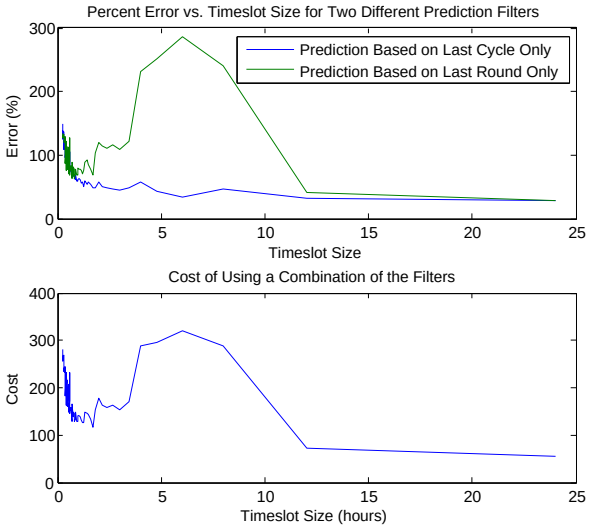



Fig. 4. Usefulness of periodic and random data for dataset 1.

the timeslot is larger. The latter concept is illustrated in Figure 3(b). In the first period shown, there is a sudden drop in solar intensity, which could be the result of a cloud. If the smaller timeslot size is used (narrower box) to characterize the average intensity for that time, then the random variation caused by the cloud will be weighed too heavily in a prediction for the same time in the second period. However, this issue can be resolved by using the larger timeslot size (wider box) to mask random variations between periods.

To determine how the usefulness of recent and past data varies with timeslot size, we studied the periodic (cycle-to-cycle) and round-to-round variations of three solar intensity datasets for several different timeslot sizes. The variations for several different timeslot sizes were calculated as follows:

$$\overline{error_r} = \frac{\sum_{i=2}^{N} \frac{|I[i]-I[i-1]|}{I[i-1]}}{N-1} \tag{6}$$

and

$$\overline{error_p} = \frac{\sum_{i=N_R+1}^{N} \frac{|I[i]-I[i-N_R]|}{I[i-N_R]}}{N-N_R}, \tag{7}$$

where $N = |I|$ and $N_R$ varied from 1 to 96. $\overline{error_r}$ is the average error between intensities in consecutive rounds and $\overline{error_p}$ is the average error beween intensities in consecutive periods. Essentially, we have predicted solar intensity based on either the last cycle only (periodic data) or the last round only (random data) and found the average prediction errors. Prediction filters in the form of (5) will use a combination of periodic and random data to infer the intensity of a future timeslot. The sum of the above error functions was computed to determine the cost (in terms of error) of using a combination of periodic and random data with varying timeslot sizes. The results are shown in Figure 4. The bottom subplot shows the error of using a combination of the filters. A local minimum is reached at approximately $\triangle t = 1h$. Similar results are obtained from the other two datasets. Therefore, we will use $N_R = 24$. This method may be improved by using a weighted average or even product of the error functions to determine the cost. However, this would yield timeslot sizes that are

optimized for a specific filter function $f(p, r)$. Here, we will only compare prediction filters with equal timeslot sizes.

## IV. Existing Filters

In this section, we will redefine the prediction filters used in the EEHF [7] and E-EEHF [8] frameworks using our formal filter specification. We will later compare these two filters to four filters developed in this work. These two filters were chosen for comparison because of their simplicity.

### A. EEHF Filter

The autoregressive prediction filter used in the environmental energy harvesting framework (EEHF) [7] can be described using our filter specification as

$$f_{EEHF} = p_{EEHF}(I, i) + r_{EEHF}(I, i), \tag{8}$$

$$p_{EEHF}(I, i) = 0, \tag{9}$$

$$r_{EEHF}(I, i) = \alpha I[i] + (1 - \alpha) r_{EEHF}(I, i - 1)], \tag{10}$$

$$N_{R_{EEHF}} = 24, \tag{11}$$

and

$$T_{EEHF} = 24 \text{ hours}. \tag{12}$$

The EEHF filter does not explicitly incorporate periodic data into its prediction. Instead, it bases its prediction solely on an exponentially-weighted moving average of random data. The weight factor $\alpha$ controls the decay rate.

### B. E-EEHF Filter

The authors of [8] have developed a prediction filter with better accuracy than the one incorporated into EEHF. Their filter is described as part of their enhanced environmental energy harvesting framework (E-EEHF). Using our filter specification, the E-EEHF prediction filter is defined as

$$f_{E-EEHF} = p_{E-EEHF}(I, i+1) + \beta r_{EEHF}(I, i+1), \tag{13}$$

$$\begin{aligned} p_{E-EEHF}(I, i) =& \alpha I[i - N_{R_{E-EEHF}}] \\ &+ (1 - \alpha) p_{E-EEHF}(I, i - N_{R_{E-EEHF}}), \end{aligned} \tag{14}$$

$$r_{E-EEHF}[i + 1] = I[i] - p_{E-EEHF}(I, i - 1), \tag{15}$$

$$N_{R_{E-EEHF}} = 24, \tag{16}$$

and

$$T_{E-EEHF} = 24 \text{ hours}. \tag{17}$$

The $\beta$ coefficient is defined as

$$\beta = \frac{I[i - N_{R_{E-EEHF}}]}{I[i - N_{R_{E-EEHF}} - 1]}. \tag{18}$$

The E-EEHF filter improves upon the EEHF filter by considering both periodic and random data.

## V. Proposed Filters

This section presents four filter designs with varying complexity and design philosophies. The Last-Round-Only (LRO) and Last-Cycle-Only (LCO) filters are the simplest base cases. The "Mixture of Cycles and Rounds" (MCR) filter combines data from previous rounds and previous cycles for better accuracy than the LRO and LCO filters. The ABC filter is based on the prediction filter utilized in the E-EEHF framework. However, it has better accuracy, and smaller energy demands.

### A. Last-Round-Only (LRO) Filter

The Last-Round-Only (LRO) filter assumes that the solar intensity at time $t$ will be close to the solar intensity at time $t - \triangle t$. We define the LRO filter as

$$f_{LRO} = p_{LRO}(I, i + 1) + r_{LRO}(I, i + 1), \tag{19}$$

$$p_{LRO}(I, i + 1) = 0, \tag{20}$$

$$r_{LRO}(I, i + 1) = I[i], \tag{21}$$

$$N_{R_{LRO}} = 24, \tag{22}$$

and

$$T_{LRO} = 24 \text{ hours}. \tag{23}$$

The LRO filter represents one extreme where only random (recent) data is used and periodic data is ignored. This type of filter is most useful when the dataset which it is applied to has a large cycle-to-cycle variance.

### B. Last-Cycle-Only (LCO) Filter

The Last-Cycle-Only (LCO) filter assumes that the light intensity at time $t$ will be close to the light intensity at time $t - N_{R_{LCO}} \triangle t$. We define the LCO filter as

$$f_{LCO} = p_{LCO}(I, i + 1) + r_{LCO}(I, i + 1), \tag{24}$$

$$p_{LCO}(I, i + 1) = I[i + 1 - N_{R_{LCO}}], \tag{25}$$

$$r_{LCO}[i + 1] = 0, \tag{26}$$

$$N_{R_{LCO}} = 24, \tag{27}$$

and

$$T_{LCO} = 24 \text{ hours}. \tag{28}$$

The LCO filter represents the opposite extreme where only periodic intensity data is considered, and random data is ignored. This type of filter is most useful when the dataset which it is applied to has a small cycle-to-cycle variance.

### C. Mixture of Cycles and Rounds (MCR) Filter

The "Mixture of Cycles and Rounds" (MCR) filter assumes that the light intensity at time $t$ will be a weighted average of intensities from recent rounds and intensities from past periods. We define the MCR filter as

$$f_{MCR} = \beta p_{MCR}(I, i+1) + (1-\beta) r_{MCR}(I, i), \quad (29)$$

$$p_{MCR}(I, i+1) = \alpha_1 p_{MCR}(I, i+1-N_{R_{MCR}}) + (1-\alpha_1) I[i], \quad (30)$$

$$r_{MCR}(I, i) = \alpha_2 r_{MCR}(I, i-1) + (1-\alpha_2) I[i], \quad (31)$$

$$N_{R_{MCR}} = 24, \quad (32)$$

and

$$T_{MCR} = 24 \text{ hours}. \quad (33)$$

The three factors $\alpha_1$, $\alpha_2$, and $\beta$ adjust how heavily past and recent data are weighted, and are between 0 and 1. Since $p_{MCR}$ and $r_{MCR}$ are exponentially-weighted moving average filters, $\alpha_1$ and $\alpha_2$ will control the rate of decay of data from past cycles and past rounds, respectively. The LRO and LCO filters are each special cases of the MCR filter when $\alpha_2 = 0$, $\beta = 0$ and $\alpha_1 = 0$, $\beta = 1$, respectively. The MCR filter is most useful when there is low round-to-round and cycle-to-cycle variance.

### D. Array of Beta Coefficients (ABC) Filter

The authors of [8] recognized that the ratio of the solar intensities in two subsequent rounds is approximately constant among different periods (days). This idea is leveraged in the ABC filter, which tracks the ratio of each round to its previous round in an array of exponentially-weighted moving averages (called betas). The ABC filter is defined as

$$f_{ABC} = p_{ABC}(I, i+1) r_{ABC}(I, i), \quad (34)$$

$$p_{ABC}(I, i) = \beta[i] = \alpha\beta[i - N_{R_{ABC}}] + (1-\alpha)\frac{I[i]}{I[i-1]}, \quad (35)$$

$$r_{ABC}(I, i) = I[i], \quad (36)$$

$$N_{R_{ABC}} = 24, \quad (37)$$

and

$$T_{ABC} = 24 \text{ hours}. \quad (38)$$

In the ABC filter, the periodic component is the beta coefficient for a particular round, and the random component is the intensity of the current round. From the definition, it can be seen that the ABC filter requires only six operations: three multiplications, a division, and two additions. The ABC filter's storage requirements depend on the number of rounds, $N_{R_{ABC}}$. In this case, since $N_{R_{ABC}} = 24$, the filter needs to store 24 beta coefficients.

TABLE I
FILTER SIMULATION RESULTS

| | Mean Prediction Error Percentage | | | |
|---|---|---|---|---|
| **Filter** | **DS1** | **DS2** | **DS3** | **Avg.** |
| **EEHF** | 85.73 | 1262.60 | 129.35 | *651.75* |
| **E-EEHF** | 68.75 | 115.72 | 51.40 | *82.49* |
| **LRO** | 80.40 | 994.62 | 105.23 | *515.91* |
| **LCO** | 60.77 | 125.19 | 75.54 | *97.50* |
| **MCR** | **60.40** | 683.35 | 90.42 | *363.23* |
| **ABC** | 70.96 | **96.60** | **50.94** | ***73.57*** |

## VI. FILTER COMPARISON METRIC

In past works, such as [7] and [8], prediction filters were indirectly compared by examining the lifetime of the network on which they were utilized. This metric depends heavily on the WSN algorithms and topology. Here, we introduce a new metric that is independent of a specific WSN and allows filters to be compared directly. At minimum, a fair metric should include the accuracy of the filter, the filter's storage requirements, the filter's calculation complexity, the filter's broadcast rate, and perhaps most importantly, the cost of the filter's misprediction.

We will define our filter metric as a cost function:

$$C_f = \omega_1 C_P + \omega_2 C_{MP} + \omega_3 C_S + \omega_4 C_C + \omega_5 C_B, \quad (39)$$

where $C_f$ is the cost of using filter $f$, $C_P$ is the cost of using the filter's prediction, $C_{MP}$ is the cost of a misprediction, $C_S$ is the cost of the filter's storage requirements, $C_C$ is the cost of the filter's calculation, and $C_B$ is the cost of the filter's prediction broadcast. The costs $C_P$ and $C_{MP}$ are related to the filter's accuracy. The costs $C_S$ and $C_C$ are related to the performance degradation caused by using the filter, as well as any extra power consumption from reading/writing memory or performing calculations. The cost $C_B$ is related to the extra power consumption and any performance degradation caused by extra radio usage when broadcasting prediction values to the rest of the WSN. The weights $\omega_n$ should be chosen such that more emphasis is given to costs that result in higher energy consumption or larger performance degradation. For example, $C_B$ should be weighed heavily because radio usage consumes a lot of power relative to the other components in a wireless sensor node. We have not derived any specific weights in this work.

In (39), $C_P \propto \overline{error_f}$, $C_{MP} \propto \frac{1}{N_R}$, $C_S \propto kN_{values}$, $C_C \propto \gamma_a A + \gamma_m M + \gamma_d D$, and $C_B \propto N_R$, where

$$\overline{error_f} = \frac{\sum_{i=1}^{N} \frac{|I[i] - I_p[i]|}{I[i]}}{N}, \quad (40)$$

$k$ is the size of a typical stored value in bits (e.g. 32 bits for integers), $N_{values}$ is the number of values that the filter needs to store, $\gamma_a$, $\gamma_m$ and $\gamma_d$ are the number of additions, multiplications, and divisions required by the filter operation, and $A$, $M$, and $D$ are the energy costs associated with additions, multiplications, and divisions, respectively.

TABLE II
FILTER COSTS

| | Sensitivity of Cost | | | | |
|---|---|---|---|---|---|
| **Filter** | **Prediction ($C_P$)** | **Misprediction ($C_{MP}$)** | **Storage ($C_S$)** | **Calculation ($C_C$)** | **Broadcast ($C_B$)** |
| **EEHF** | 651.7522 | $\frac{1}{N_R}$ | $2k$ | $2M + 2A$ | $N_R$ |
| **E-EEHF** | 82.4898 | $\frac{1}{N_R}$ | $2k(N_R + 1)$ | $3M + 1D + 4A$ | $N_R$ |
| **LRO** | 515.9050 | $\frac{1}{N_R}$ | $k$ | 0 | $N_R$ |
| **LCO** | 97.4965 | $\frac{1}{N_R}$ | $kN_R$ | 0 | $N_R$ |
| **MCR** | 363.2256 | $\frac{1}{N_R}$ | $k(N_R + 4)$ | $6M + 6A$ | $N_R$ |
| **ABC** | 73.5685 | $\frac{1}{N_R}$ | $kN_R$ | $3M + 1D + 2A$ | $N_R$ |

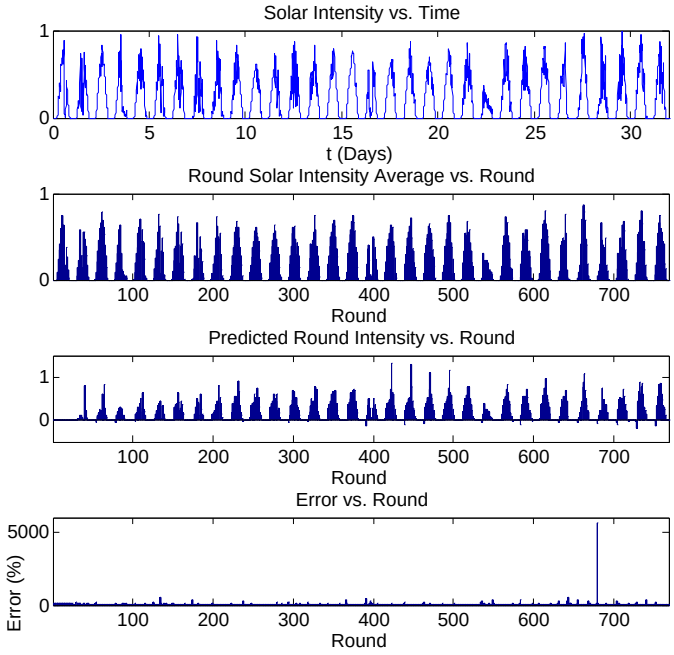



Fig. 5. Simulation run of the ABC filter on dataset 3.

## VII. RESULTS

The six filters defined in Section IV and Section V were simulated on three datasets. The first dataset (DS1) is from the data published in [7], and the other two (DS2 and DS3) are composed of data collected at RIT. As specified in their definitions, each filter utilized 24 rounds per period. For the EEHF filter, $\alpha = 0.9$. For the E-EEHF filter, $\alpha = 0.5$. For the MCR filter, $\alpha_1 = 0.9$, $\alpha_2 = 0.1$, and $\beta = 0.5$. For the ABC filter, $\alpha = 0.9$. Figure 5 shows an example simulation run for the ABC filter with DS3. The top subplot is the actual intensity data from RIT over a 32-day period with samples taken every ten minutes. The next subplot is a bargraph of the average intensity in each round. Since $N_R = 24$, each bar is an hourly average. The third subplot shows the ABC filter's intensity prediction for each round, and the final subplot shows the relative error between the prediction and the actual average intensity. There is a very large error between round 600 and round 700 that would most likely be calculated as an outlier. To be fair, however, these data were not removed from any of the filter results, as they could represent mispredictions that cause a node to be completely drained of its residual energy. Simulations like the one shown in Figure 5 were run for each of the six filters and each of the three datasets, resulting in 18 total simulations.

Table I summarizes the results. For DS1, the MCR filter had the lowest mean prediction error. The ABC filter had the lowest mean prediction error for datasets 2 and 3. The final average in the last column is a weighted average with weights equal to the number of days in each dataset divided by the sum of the number of days across all three datasets. The E-EEHF and ABC filters, on average, have better accuracy than the other filters. The high error percentages show the difficulty in making highly accurate predictions with low-complexity filters, a fact that should be considered in the design of other WSN components. Table II summarizes sensitivity of the costs associated with each filter; each cost is proportional to the given factor(s). The ABC filter requires about half of the storage required for the E-EEHF filter and also has smaller calculation and prediction costs.

## VIII. CONCLUSIONS

In this work we have analyzed the characteristics of typical solar intensity data. Using those characteristics, we developed a general form for a solar intensity prediction filter. Since solar panel electrical current output is proportional to solar intensity, the filters can be used to predict the energy harvesting capabilities of solar-powered wireless sensor nodes. A comparison metric was also developed and used to compare two existing and four proposed filters based on their storage requirements, calculation complexity, and accuracy. Results show that the Array of Beta Coefficients (ABC) filter is less expensive in terms of computation and storage than all other analyzed filters. It also has an ∼8-fold improvement in accuracy over the EEHF filter.

## IX. ACKNOWLEDGMENTS

The authors wish to acknowledge Richard Stein and RIT Facilities Management Services for the collection of solar intensity data.

## REFERENCES


[1] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 18–25, March-April 2006.

[2] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. New York, NY, USA: ACM, 2002, pp. 88–97.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393 – 422, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/B6VRG-44W46D4-1/2/f18cba34a1b0407e24e97fa7918cdfdc

[4] D. Brunelli, L. Benini, C. Moser, and L. Thiele, "An efficient solar energy harvester for wireless sensor nodes," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008, pp. 104–109.

[5] X. Jiang, J. Polastre, and D. Culler, "Perpetual environmentally powered sensor networks," in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, April 2005, pp. 463–468.

[6] F. Simjee and P. H. Chou, "Everlast: long-life, supercapacitor-operated wireless sensor node," in *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2006, pp. 197–202.

[7] A. Kansal and M. B. Srivastava, "An environmental energy harvesting framework for sensor networks," in *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2003, pp. 481–486.

[8] K. Kinoshita, T. Okazaki, H. Tode, and K. Murakami, "A data gathering scheme for environmental energy-based wireless sensor networks," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, Jan. 2008, pp. 719–723.

# An Ultra Low Power Digitally Controlled Oscillator with low jitter and high resolution

Nasser Erfani Majd, Mojtaba Lotfizad, Arash Abadian
Department of Electrical and Computer Engineering
Tarbiat Modares University (TMU)
Tehran, Iran
Email: n.alboghobiesh@modares.ac.ir
lotfizad@modares.ac.ir
abadian@modares.ac.ir

Mohammad Bagher Ghaznavi Ghoushchi
Department of engineering
Shahed University
Tehran, Iran
Email: Ghaznavi AT shahed.ac.ir

*Abstract*— **In this paper, an ultra low power and low jitter 12bit CMOS digitally controlled oscillator (DCO) design is presented. Based on a ring oscillator implemented with low power Schmitt trigger based inverters. Simulation of the proposed DCO using 32nm CMOS Predictive Transistor Model (PTM) achieves controllable frequency range of 550MHz~830MHz with a wide linearity and high resolution. Monte Carlo simulation demonstrates that the time-period jitter due to random power supply fluctuation is under 31ps and the power consumption is 0.5677mW at 750MHz with 1.2V power supply and 0.53-ps resolution. The proposed DCO has a good robustness to voltage and temperature variations and better linearity comparing to the conventional design.**

***Keywords- digitally controlled oscillator (DCO); low power; jitter; linearity; robust;***

## I. INTRODUCTION

PHASE-LOCKED loops (PLLs) are widely used in many communication systems to clock and data recovery or frequency synthesis [1]. Typical analog PLLs include a phase-frequency detector, a charge pump, a loop filter, a voltage controlled or current controlled oscillator, and a frequency divider [2, 3]. The controlled oscillator is the key component in the core of PLL. Recently, efforts have been made toward the development of fully digital PLLs. Compared to their analog counterparts, fully digital PLLs exhibit better noise immunity and they are invulnerable to DC offset and drift phenomena [4, 5, 6]. Digitally controlled oscillator (DCO) is a replacement of the conventional voltage or current controlled oscillator in the fully digital PLLs. DCO is the heart of the ADPLL that shows higher noise immunity and robustness than the conventional PLLs [1]. DCO dominates the major performances of ADPLL such as power consumption and jitter, and hence is the most important component of such clocking circuits [4, 6, 7]. Since DCO occupies 50% power consumption of an ADPLL [7], the power consumption of DCO should be reduced further to save overall power dissipation to meet low power demands in SOC designs.

The Block diagram of the ring oscillator based DCO which is used in this paper is shown in Fig 1. It consists of digitally controlled delay elements which are controlled by coarse and fine bits and a control logic block for enabling DCO and

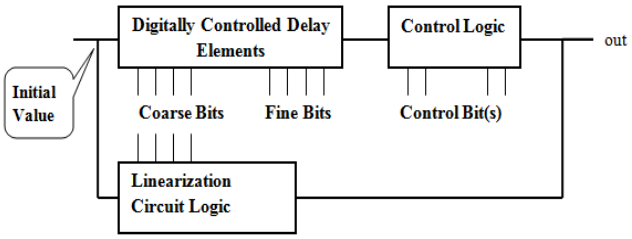



Figure 1.Block diagram of the ring oscillator DCO

linearization circuit for linearizing the DCO period by increasing the input code. DCO starts to work by applying the initial value to the circuit.

Basically, two main techniques exist for designing the DCO as shown in fig. 2. One technique changes the MOS driving strength dynamically using a fixed capacitance loading and achieves a fine resolution [8, 9]. While the other uses shunt capacitor technique to tune the capacitance loading [10, 11]. They both have good linear frequency response and a reasonable frequency operating range. Power consumption is an important problem for portable battery charged computing systems, so the reduction of the power consumption has become a major concern.

A simple DCO that directly uses an inverter ring is presented in [12], but has insufficient resolution for most applications. Another DCO example consists of bank of tri-state inverter buffers [13]. The delay resolution in this case can be controlled by the number of enable buffers. However, [13] has the disadvantages of large silicon area and high power consumption. Another means of fine resolution enhancement, implemented by an Or-And-Inverter (OAI) cell shunted with two tri-state inverters to enhance driving capability, was

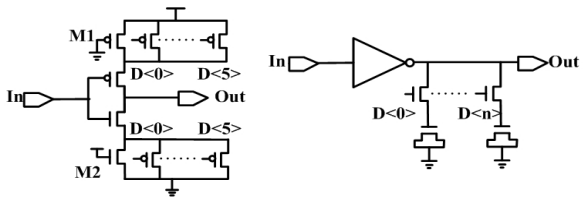



Figure 2.Standard Cell of Digitally controlled oscillator. (a) Driving strength controlled. (b) Shunt capacitance controlled.

proposed in [3]. The proposed DCO in [3] has less area and power consumption than [13]. However, the resolution step of the proposed DCO is nonuniform and sensitive to power-supply variation because it uses OAI cell to change the delay resolution, this technique also requires an additional decoder for mapping OAI cell control input.

This paper presents a low power, low jitter and high resolution DCO using binary controlled pass transistors and low power Schmitt trigger. The DCO is designed using the 32nm CMOS Predictive Transistor Model (PTM) and HSPICE simulator.

## II. Conventional And Proposed Dco Architecture

DCO should generate an oscillation period of $T_{DCO}$, which is a function of digital input word D and given by:

$$T_{DCO} = f(d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \cdots + d_1 2^1 + d_0 2^0) \quad (1)$$

the DCO transfer function is defined such that the period of oscillation $T_{DCO}$ is linearly proportional to digital word D with an offset

$$T_{DCO} = T_{offset} - D.T_{step}, \text{ D:Digital control bits} \quad (2)$$

where $T_{offset}$ is a constant offset period and $T_{step}$ is the period of quantization step. For the conventional driving strength controlled DCO shown in Fig. 3, the delay tuning range of this standard cell is obtained as follows:

$$\frac{T_{tune}}{2} = (C_1 + C_2)(R_1 // \frac{\Delta R}{d_0} // \frac{\Delta R}{d_1 2} // \cdots // \frac{\Delta R}{d_{n-1}2^{n-1}}) - (C_1 + C_2)R_1 \quad (3)$$

$$= \frac{R_1(C_1+C_2)}{1+(D\cdot\Delta W)/W_1} - (C_1+C_2)R_1 \quad (4)$$

$$\approx R_1(C_1+C_2)\frac{D\cdot\Delta W}{W_1}, \quad (\text{Only if } \frac{D\cdot\Delta W}{W_1} << 1) \quad (5)$$

where $R_1$ is the equivalent resistance of M1 and W1 is the width of M1. In order to have a good linear tuning range, the width of transistor M1 has to be increased as can be seen in Equation (5). Consequently, the equivalent resistance $R_1$ will decrease resulting in a smaller delay tuning range. One way to increase the tuning range while keeping the linear response is to increase the capacitance loading. However this will minimize the maximum frequency that the DCO can accomplish and the power consumption will also be increased.

The proposed DCO is based on ring oscillator implemented with low power Schmitt trigger based inverters. It uses binary controlled pass transistor arrays to control the period of DCO. Schmitt trigger based inverter has a higher low to high switching threshold and lower high to low switching threshold compared to the conventional As a result, the proposed DCO circuit provides the same tuning range with smaller capacitance loading, which is beneficial for power consumption reduction. Moreover, in conventional DCO circuit, the slope of the input signal to each stage decreases gradually due to the large delay between each stage. This

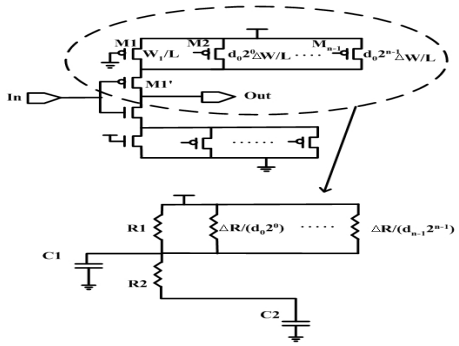



Figure 3. Equivalent circuit for the calculation of delay tuning range.

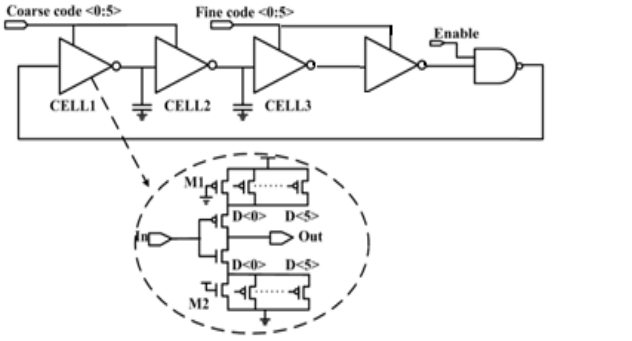
result in not only non-ideal rail-to-rail switch but also a poor power performance. The steep slope of the output signal from the Schmitt trigger based inverter minimizes this problem to certain extend. The improved DCO has two coarse delay cells and two fine delay cells and a NAND gate for reset. We don't use the Schmitt trigger in fine delay cells of DCO, because Schmitt trigger transistors are switched in each cycle, so they themselves consume a lot of power in the DCO therefore omission of Schmitt trigger from fine delay cells can decrease the power consumption of the circuit. Since fine delay cells of DCO do not have a capacitance loading thus fine delay cells output signal is still sharp and omission of the Schmitt trigger from fine delay cells does not disturb the DCO performance. Furthermore we use the low power Schmitt trigger in coarse delay cells which has two inverters in its structure and these inverters act as buffers in the signal path and by reconstructing the signal, reduce the jitter of the DCO. This Schmitt trigger is reported in [14]. The high to low and low to high switching threshold of the Schmitt trigger is obtained as follows

$$V_{TR} = \frac{Vtn + \sqrt{Kn/Kp}\,(VDD - Vtp)}{1 + \sqrt{(\frac{Kn}{Kp})}} \quad (6)$$

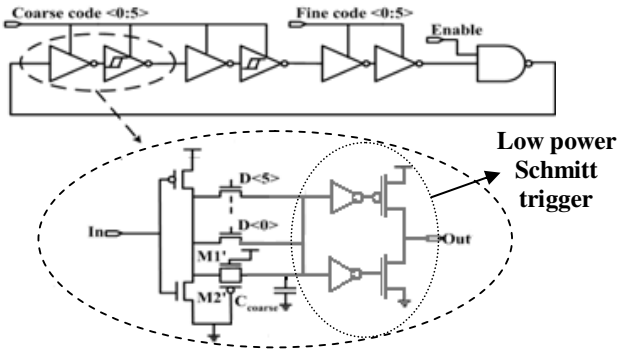
Where Kn and Kp are the transconductance factors of $Mn_{inv}$ and $Mp_{inv,}$ and Vtn and Vtp are their respective threshold voltages. The circuit diagrams of the conventional DCO and proposed DCO are showed in Fig. 4. In order to compare the power consumption, both circuits must be equally sized.

## III. Comparsion Of Power Consumption Between The Conventional And Proposed Dco Structures

Two structures of DCO are simulated and compared using 32nm CMOS PTM (Predictive Transistor Model) with a supply voltage of 1.2Volts and HSPICE simulator. The impact of each control bit on the period of the two DCO structures is shown in table 1. Both structures have the same linear tuning range until $5^{th}$ bit is asserted. This is due to the fact that the requirement for linear tuning range fails when D.ΔW becomes too large comparing to W1 so these structures are linear for the first 32 input coarse codes. In order to compare the power consumption, the first 5 control bits are chosen instead of 6, since the last control bits contributes to non-linear tuning range




(a)




(b)

Figure 4. Digitally Controlled Oscillator. (a) Conventional DCO structure, (b) proposed DCO structure.

which is not desired for DCO. Moreover, since two DCO structures have the same operation ranges, it is more reasonable for us to compare their power consumption. Compared to the conventional DCO, the proposed DCO saves approximately 70% power consumption as shown in fig 5. As discussed in section II, this reduction is due to the comparatively smaller capacitance loading for the Schmitt trigger based inverter than the conventional inverter at the same operating frequency and the using the low power Schmitt trigger in the inverters of coarse delay cells. The proposed DCO is more power efficient than the conventional DCO.

TABLE I. Impact Of Each Control Bit On The DCO Period

| Control bits | Conventional DCO | | Proposed DCO | |
|---|---|---|---|---|
| | Period (ns) | Delta (ps) | Period (ns) | Delta (ps) |
| 100000 | 1.1905 | 246.7 | 1.1902 | 242.8 |
| 010000 | 1.4372 | 124.5 | 1.4330 | 134.8 |
| 001000 | 1.5617 | 84.5 | 1.5678 | 91.6 |
| 000100 | 1.6462 | 42 | 1.6594 | 32.7 |
| 000010 | 1.6882 | 20.1 | 1.6921 | 18.1 |
| 000001 | 1.7083 | 21.9 | 1.7102 | 19.9 |
| 000000 | 1.7302 | - | 1.7301 | - |

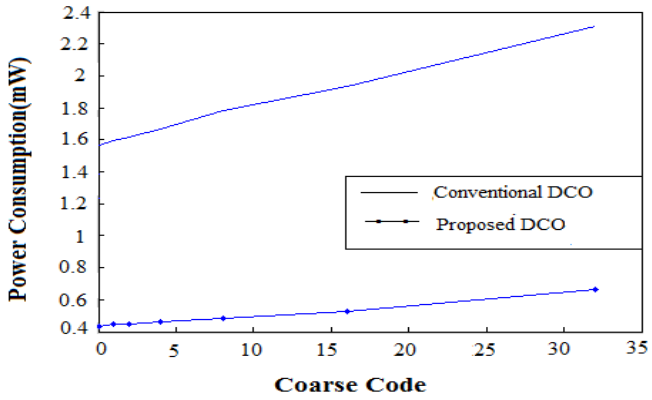



Figure 5. Power consumption of the two DCO structures

## IV. Improving Linear Operating Range Of the Proposed DCO And Simulation Results

The proposed DCO which is explained in section II has a limited linear operating range as discussed above. In this paper , three stage constant delay chains and 4:1 multiplexer are used to increase the operating range, the proposed DCO and its linearization circuit are shown in Fig. 6. The 6th bit is taken off for better linear response and the 1st bit is also taken off for larger coarse resolution. So this structure is linear for 64 input coarse codes instead of 32 input coarse codes. The proposed DCO structure with increased operating range is designed and simulated using 32nm CMOS PTM model and HSPICE simulator. The frequency ranges of the coarse and fine tuning loops are shown in Fig. 7. The curves have a good linearity which is a key factor of the PLL performance. The operational frequency response to the process, temperature and voltage variations are shown in Fig. 8. The curves show the normalized data with respect to the center frequency. Fig. 8 shows that the relative delay per code is almost the same regardless of the process, temperature and voltage variations, which means this DCO design is robust to PVT variations. We can extend the linearization circuit to achieve a 14-bit DCO which is linear for 128 input coarse codes. Extended linearization circuit is shown in Fig. 9. It consists of a seven stage constant delay chain and 8:1 Mux. The 7th and 6th bits are taken off for better linear response and 1st bit is also taken off for better coarse resolution. The simulation results show that the DCO curve has a good linearity. The frequency ranges of the coarse tuning loop are shown in Fig. 10.

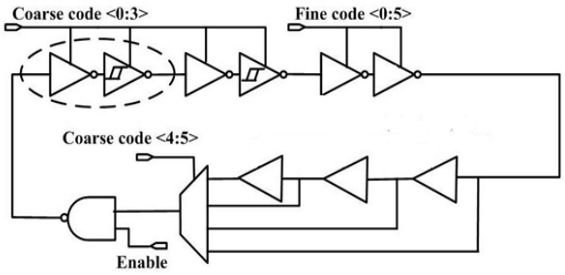



Figure 6. The proposed DCO structure with improved linear operating range

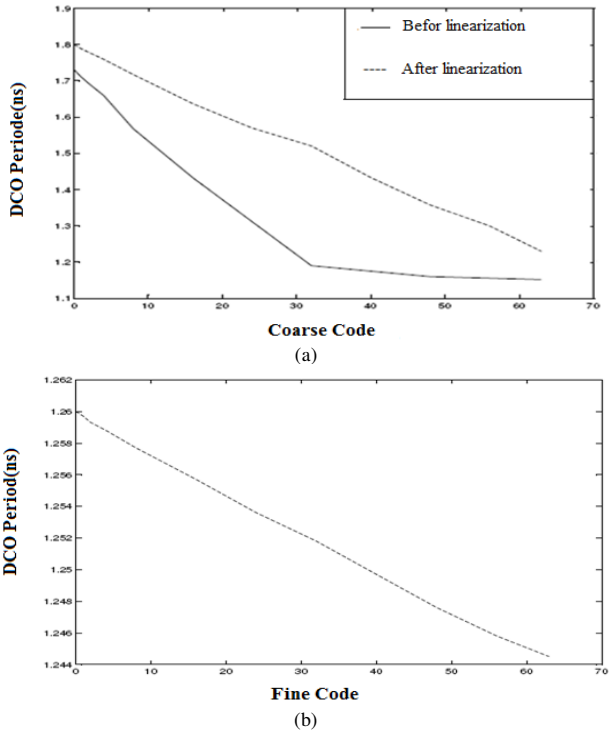



Figure 7. Operating range of the proposed DCO. (a) Coarse loop (b) Fine loop

TABLE II. Characteristic Of The Proposed DCO

| Items | Coarse delay | Fine delay |
|---|---|---|
| Resolution | 6 bit | 6bit |
| Max. DCO Gain | 13ps | 0.53ps |
| Avg. DCO Gain | 9ps | 0.25ps |
| Operation range | 550~830 MHz | |
| Operation range | 0.5677mW @ 750 MHz | |

The time-period jitter is the time difference between the measured cycle period and the ideal cycle period. The jitter performance of the proposed DCO is simulated by Monte Carlo analysis using a Gaussian distribution function taking into account 10% variation in supply voltage. The results are shown in Fig. 11 by overlapping every cycle period. A 31ps time-period Jitter is measured.

Table 3 shows the measurement results to compare with a few recent state-of-the-arts DCO designs [1, 3, 10, 13]. The proposed DCO achieves the finest LSB resolution and the highest operating frequency. In addition, the proposed DCO consumes less power than the others.

## V. CONCLUOSION

A low power 12 bit digitally controlled CMOS oscillator (DCO) design for low power consumption and low jitter are presented. The presented DCO demonstrate a good robustness to voltage and temperature variations and better linearity comparing to the conventional design. Simulation of the proposed DCOs using 32 nm CMOS Predictive Transistor Model and HSPICE simulator achieves a frequency of 550~830 MHz and power consumption of 0.5677mW at 750 MHz and 1.2V power supply. The performance, flexibility, and robustness make the proposed DCO viable for high performance fully digital PLL application.

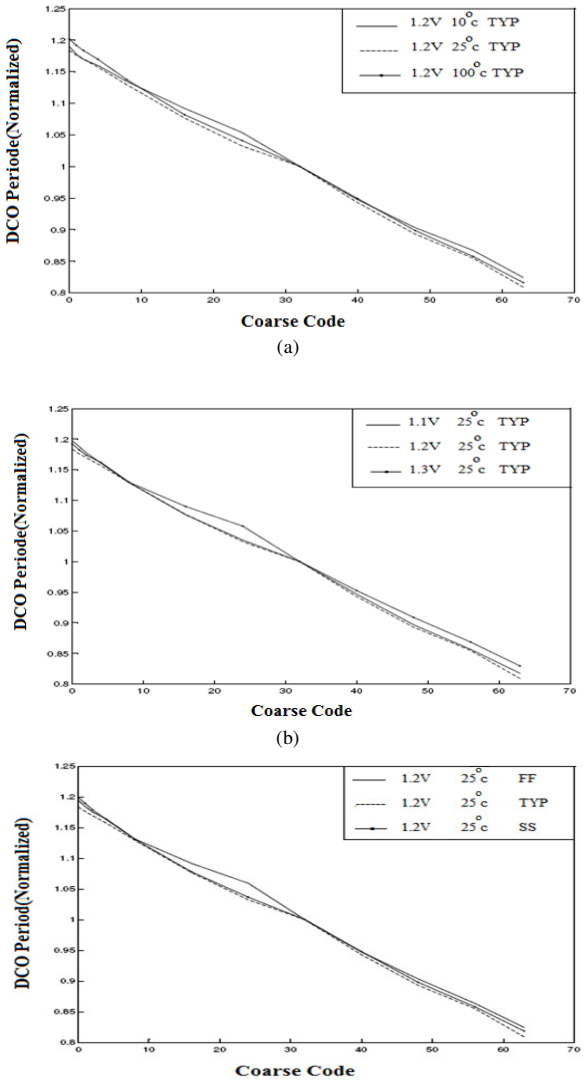



Figure 8. Delay characteristics of the coarse loop according to Process, Voltage and Temperature variations. (a) Temperature Variation, (b) Voltage Variation, (c) Process Variation

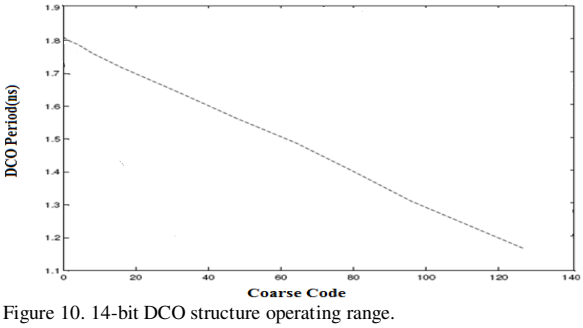



Figure 10. 14-bit DCO structure operating range.

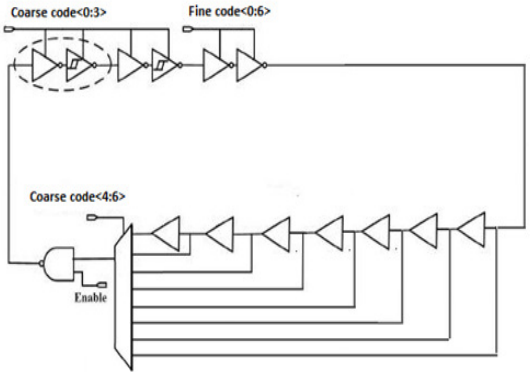



Figure 9. 14-bit proposed DCO structure

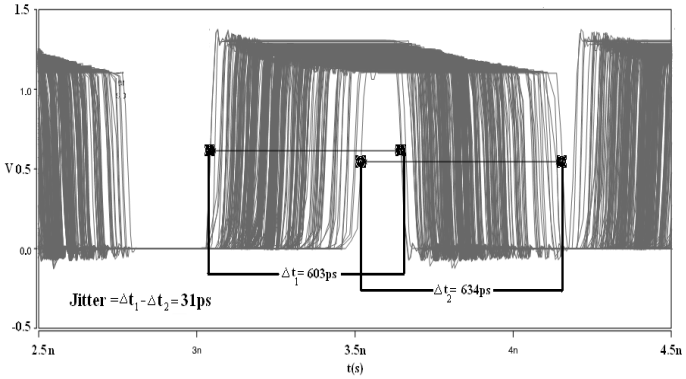



Figure 11. Time-period jitter of the proposed DCO (Monte Carlo analysis)

TABLE III. COMPARISON WITH EXISTING DCOS.

| Function | [1] | [10] | [13] | [4] | Proposed DCO |
|---|---|---|---|---|---|
| Process | 0.35um @ 3.3V | 0.13 um @ 1.65V | 0.6 um @ 5V | 0.35 um @ 3.3V | 32nm@ 1.2V |
| DCO control word length | 15 bits | 8bits | 10 bits | 12 bits | 12 bits |
| Coarse Resolution | 385 ps | - | 550 ps | 300 ps | 12 ps |
| LSB(Fine) Resolution | 1.55 ps | 40 ps | 10 ps | 5ps | 0.53 ps |
| DCO output frequency | 18 ~214 (MHz) | 150(MHz) | 10 ~12.5 (MHz) | 45~450 (MHz) | 550 ~830 (MHz) |
| Power consumption | 18mW@ 200MHz | 1mW @ 150MHz | 164mW @ 100MHz | 100 mW @ 450 MHz | 0.5677mW@ 750MHz |


[1] P.-L. Chen, C.-C. Chung, and C.-Y. Lee, "A portable digitally controlled oscillator using novel varactors," *IEEE Transactions on Circuits and Systems II*, vol. 52, no. 5, pp. 233–237, 2005.

[2] Roland E. Best: "Phase-locked loops. Theory, Design, and applications," McGraw-Hill Book Company, 1984.

[3] B. Razavi, "Monolithic Phase-Locked Loops and Clock-Recovery Circuits," IEEE Press, 1996. Collection of IEEE PLL papers.

[4] C. Chung and C. Lee, "An all-digital phase-locked loop for high-speed clock generation," IEEE J. Solid-State Circuits, vol. 38, pp. 347–351, Feb. 2003

[5] P. Nilsson and M. Torkelson, "A monolithic digital clock-generator for on-chip clocking of custom DSP's," IEEE J. Solid-State Circuits, vol.31, pp. 700–706, May 1996

[6] J. Dunning, G. Garcia, J. Lundberg, and E. Nuckolls, "An all-digital phase-locked loop with 50-cycle lock time suitable for high performance microprocessors," IEEE J. Solid-State Circuits, vol. 30, pp. 412–422, Apr. 1995.

[7] T.Olsson and P.nilsson, "A digitally controlled PLL for SoC application," IEEE j. Solid-state Circuits, Vol.39, no. 5,pp.751-760, May 2004.

[8] R. B. Staszewski and P. T. Balsar, " Phase-Domain All-Digital Phase- Locked Loop,"IEEE Trans. Circuits and Systems II, Vol. 52, pp. 159- 163, Mar. 2005.

[9] M. Saint-Laurent et al, "A Digitally Controlled Oscillator Constructed Using Adjustable Resistor," IEEE Southwest Symposium on Mixed- Signal Design, 2001.

[10] P. Raha, S. Randall, R. Jennings, B. Helmick, A. Amerasekera, and B.Haroun, "A robust digital delay line architecture in a 0.13-_m CMOS technology node for reduced design and process sensitivities," in Proc. ISQED'02, pp. 148–153, Mar. 2002.

[11] P. Andreani, F. Bigongiari, R Roncella, R. Saletti and P.Tenini, "A Digitally Controlled Shunt Capacitor CMOS Delay Line," Analog Circuits and Signal Processing, Kluwer Academic Publishers, Volume 18, pp. 89-96. 1999.

[12] T. Olsson and P. Nilsson, "Portable digital clock generator for digital signal processing applications," *Electron. Lett.*, vol. 39, pp. 1372–1374, Sep. 2003.

[13] E. Roth, M. Thalmann, N. Felber, and W. Fichtner, "A delay-line based DCO for multimedia applications using digital standard cells only," in *Dig. Tech. Papers ISSCC'03*, Feb. 2003, pp. 432–433.

[14] V.A. Pedroni," Low-voltage high-speed Schmitt trigger and compact window comparator," Electronics Letters, vol. 41 no. 22, Oct 2005.

# Early Stage Trade-offs Analysis in Reconfigurable H.264 Video Design

Youngsoo Kim
North Carolina State University
Dept. of Electrical Engineering
Raleigh, NC, 27695 USA
youngsoo_kim@ncsu.edu

Kyungsu Kim, Seongmo Park
Electronics and Telecommunications Research Institute(ETRI)
SoC Research Department
Daejon, South Korea 305-700
{kimks0326, smpark}@etri.re.kr

## Abstract

*Current video compression algorithms such as H.264 and MPEG are increasingly complicated and difficult to analyze and profile. Although the tools and system level languages speed up the design process, they often prove to be inefficient and incapable of providing complexity analysis as a first step aiming at the implementation of video compression algorithms. The proposed profiling framework will help to develop a methodology that facilitates the derivation of analytical models. The framework proposes analytical CAL models for quantifying the underlying algorithm's memory complexity, related timing considerations, and verification of the correctness of the video compression algorithm. The methodology has been validated by applying it to an H.264 motion estimation algorithm. The experimental results present a speed 7x faster than required for assessing design metrics compared to conventional methodologies.*

## 1. Introduction

Presently system engineers begin their design with C/C++ which are difficult to assess for hardware designers. Design choices, including algorithm-architecture selection, must be determined in the very early stages of design. There is a need to reduce duplication of effort from algorithm-architecture selections for specific application scenarios within the global tradeoffs context. In consideration of these observations, a conflicting environment for designers is presented. Additionally, there exists a wide cognitive gap between the algorithm designer and the hardware/software designers whose role is to determine the hardware/software architecture for implementation of these algorithms. This gap prohibits estimation of finally implemented performance values at high levels of abstraction. In particular, hardware designers face challenges when deciding on implementation choices for a specific algorithm since the resulting performance values are not available. In short, to help designers, a profiling methodology should provide early analysis features of final hardware, and should bridge the system level-implemented hardware gap by providing complexity metrics.

In this work, we propose an analytical profiling framework for memory related costs based on reconfigurable CAL modeling. The overall design trade-off metrics are calculated interactively with the consideration of memory complexities and performance parameters. The following contributions are made in this work.

- Analytical Models for Trade-off analysis: An analytical analysis between input parameters and performance parameters is presented based on CAL modeling.
- Trade-offs Analysis of video compression algorithms: We use the proposed profiling framework to investigate the cost of hardware implementation as well as memory related costs and performance values.

This paper is organized as follows. Section 1.1 describes state-of-the-art literature survey relevant to this work. In section 2, we present video compression applications as well as CAL modeling for the framework. Additionally, this section deals with analytical modeling methodology. Section 3 provides preliminary results and comparisons with conventional methods in terms of design metrics.

### 1.1. Relevant Work

There are several tools for traditional profiling including software profiling. The basic idea of these tools is that applications spend a large share of execution time in a kernel or inner loop. Intel VTune and GNU Prof are the standard tools for this purpose but they are focused more on instruction level complexity in a program rather than on a potential measure for the final implemented system in terms of memory complexity or other timing considerations [1]. Based on open literature survey, these common tools do not provide customized design metrics such as memory complexity

and bandwidth information beyond memory access counts for hardware implementation alternatives selection. Therefore, designers are hesitant to utilize these tools to assess algorithm candidates or to provide to the architecture candidates for the application. Hardware/software co-design and compiler groups' focus has been directed toward estimating the early performance of applications. The work of HW/SW codesign tools such as Ptolemy II and Synopsys CoWARE tools leads to effective design environments which co-simulate and/or co-synthesize heterogeneous systems and techniques for optimizing and reducing memory requirements [2][3][4][5]. However, these tools rely on dynamic simulation with incremental refinement. They focus on more accurate memory metrics with the latest-possible algorithm-architecture binding. Compiler directed approaches solve this problem by providing the instruction level complexity analysis of C/C++ references and by providing semi-static information [6][7][8]. The estimations produced by the compiler and profiler depend on a specific general purpose processor platform, lacking representative metrics for custom hardware or hardware/software systems design.

## 2. Proposed Profiling Framework

We will present a case study of H.264, using our profiling framework flow. Before discussing the profiling framework methodology, let us briefly summarize the H.264 standard which is the video encoding/decoding standard that replaces the current MPEG4 video standard.

### 2.1. H.264 Video Compression Algorithm

In its H.264 video coding layer, some of the important enhancements include the use of a small block-size, an exact match transform, adaptive in-loop deblocking filter and motion prediction capabilities. A typical decoding process begins with entropy decoding. After receiving the data from the NAL (Network Adaptation Layer), the data are processed by the entropy decoder. Next, the IT/IQ (Integer transform/Inverse quantization) block is used to generate the reference frame data which will be added to the reference frame image or intra-predicted image based on its header information. Then, the original image is reconstructed through the deblocking filter. The reference C source code is built by pruning the H.264 standard C model originally from the H.264 standard committee. Extra functionality beyond the selected H.264 profile was removed from C code. Therefore, the C model that we use has been optimized at the source code level by designers. This C reference model plays the role of providing the specification and test streams for hardware implementation of H.264.

### 2.2. Reconfigurable CAL Framework

It is our observation that video coding algorithm models typically involve two stream memories and control conditions with regards to those stream memories. In this context, Petri nets are good candidates for studying and analyzing the behavior of video compression models for early estimation. Petri net is the one of models which represent interacting concurrent components and used as a design and analysis tool of systems. To implement this model and modeling in the framework, CAL has best modeling capability since it comes with language property and automated tool sets [9]. CAL Actor Dataflow Language has been chosen as the analytical modeling language for the operation of the Function Units(FUs) in the analytical model library. CAL was initially developed as a specification for the Ptolemy project [9]. RVC-CAL is a subset of the original CAL language and is being used for developing the RVC standard Video Tool Library by the IET working group.

CAL actor language can describe algorithms with interacting actors. Each of actors has its own state and functions. Communications and interactions among actors can be processed through channels or FIFOs. Actors define functions described by a set of actions. Actions typically consume input tokens, generate output tokens and modify the internal states which are very similar to Petri net's nature. CAL has expression capabilities sufficient to specify a wide range of video compression algorithms that follow a variety of dataflow models.

The proposed framework is the development of analytical models which are able to model the underlying algorithms' memory complexity in the very early design stage for video compression applications. Furthermore, it is necessary to develop a methodology that facilitates the analytical models' derivation process and verifies the accuracy of models for video algorithms. Starting from an application written in CAL behavioral description, annotations will be made with a pre-processor to embed input parameters in algorithm code. The fundamental equations for basic processing (e.g. data fetching operations and elements processing) in an algorithm (e.g. motion estimation) is derived and CAL equation libraries will be built for reusability. The objective is to offer a tool which contains a relevant basic set of CAL equations from existing algorithms. Designers who want to build extended equations for different encoding/decoding algorithms can benefit by reusing these analytical framework models. Therefore, the profiler is able to deliver a reduction of system-RTL(Register Transfer Level) level gaps with rapid estimation times. Fig. 1 presents an overall view of the design flow which is employed in a H.264 design space exploration loop. A CAL description will be defined to model an application instance. The description includes input parameters including maximum width of frame size in macroblock, size in bits of macroblock, horizontal and ver-

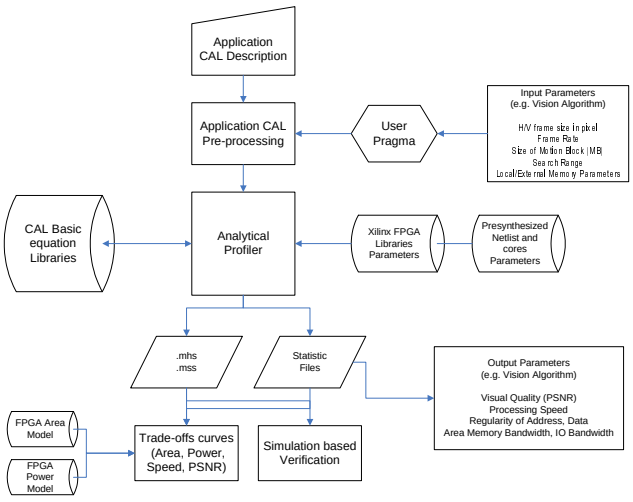


Figure 1. Design Flow for Reconfigurable CAL Framework

tical search range in pixels, etc., in H.264 algorithm. The extended sets of CAL model equations will be generated in an automated manner by the profiler, user pragma compiler. Input parameters are parsed by the utility compiler and generate extended CAL model equations by referring to user defined parameters. For example, based on the distance criteria in motion estimation algorithm, different sets of equations for blocks will be generated (e.g. Sum of Absolute Difference (SAD), Mean Absolute Error (MAE), Minimized Maximum Error (MME), Cross Correlation Function (CCF), etc.)[10]. This will allow designers to reuse an analytical model for different video algorithms and will provide designers with trade-off curves based upon design metrics (e.g visual quality (PSNR), memory bandwidth, I/O bandwidth, regularity of address and data, etc.). Trade-off curves with design space exploration statistics files will be generated for designers to assess different algorithms for different architectures in the early stage of design. This design methodology accomplishes reuse of CAL equation libraries by expanding basic CAL model sets. Basic CAL models are described using an atomic action such as data fetching or pixel processing. In addition to atomic mark up, extended CAL models are composed of one or more combined atomic CAL basic models which are derived from algorithms. The key feature of the methodology is the generation of memory complexity metrics for designers based on input parameters. Memory complexity trade-offs among those factors will help designers achieve an efficient compromise among design alternatives. We illustrate the framework with a Motion Estimation (ME) case study. Beginning with high level CAL specifications, a designer must choose a candidate architecture with reference to the ultimate design implementation.

The core problem here is to evaluate ME algorithms in order to have an idea of memory complexity. The proposed design flow begins with building basic CAL model equations. The cost of LRTB (Left-Right-Top-Bottom) accessing an image pixel in terms of the number of memory accesses can be computed as a function of window memory(W) and frame memory(M) assuming an 8x8 pixel block. Consider now that the cost of meandering access of an image pixel in terms of the number of memory accesses can be similarly computed. Memory fetching must occur when direction changes to the opposite. Meandering does not have to fill window memory in contrast to the LRTB method. In short, it will be clear that the meandering access has a lower memory bandwidth because the meandering method requires fewer refills than LRTB method based CAL atomic models.

These basic CAL model equations can be used to compute the associated costs for processing an image of a given size in the following way. When the two memory schemes are used with ME blocks in a video algorithm, the memory complexity metric must be offered to designers in order to select memory access schemes as well as motion estimation schemes with configuration parameters. To see the memory access cost for distance criteria in ME, the following extended equation (1) can be built by using basic sets of equations with the help of the framework.

$$NA_{ME} = 2 * N(ME_{scheme}) * NA(Mem_{scheme}) * Frame_{rate} \quad (1)$$

We use this extended equation to compute memory access metrics by changing N(MEScheme), NA(MemScheme), and FrameRate. Suppose that N(MEScheme) is defined by motion estimation schemes, e.g. 1:Full pel ME, 0.5:Half pel ME, and NA(MemScheme) as defined by basic CAL models. Configuration parameters such as FrameRate, and horizontal and vertical image size are values provided by the user and the framework. Exploration trade-off curves which include a memory complexity metric against DSE parameters is presented to designers.

## 3. Preliminary Results

This profiling design flow begins by building basic CAL models. These basic CAL equations are used to compute the associated costs for processing memory fetch operation. In order to see the computational complexity cost for operations in overall motion estimation, the extended equation can be built by using basic sets of equations with the help of the profiler. Exploration trade-off curves which include a customized complexity metric with various output parameters are presented to designers. Figure 2 shows results of design space exploration curves of H.264 encoding in terms of Quantization Parameters(QP) and reconstructed image frame quality. This also shows Peak Signal to Noise

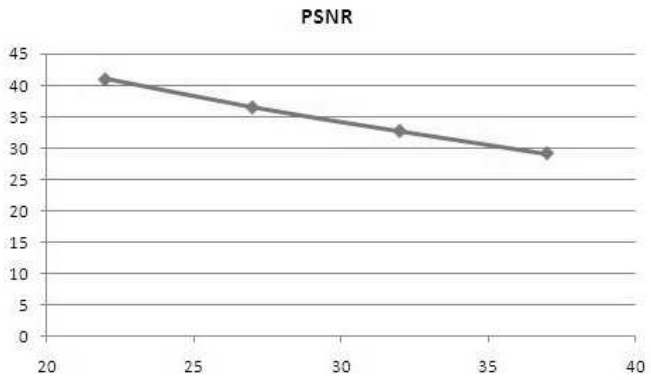



Figure 2. DSE curve of PSNR vs. QP

Table 1. Early Estimation TAT Reduction

| | HDL modeling | This framework |
|---|---|---|
| Estimation Time | 4 weeks | 3 days |
| Lines of code | 3,900(verilog) | 920 |

Ratio(PSNR) in variations of QPs. Table 1 confirms the effect of use of analytical models for early estimation gain. Using equation based models reduces Turn Around Time (TAT) of building simulation models and performance estimation time. The time for modifying simulation model description and simulating it grows in a linear manner with reference to the size of the description. Also, it is time consuming and error prone to build test benches and verification scripts to run conventional simulation models. The results show the advantage of using CAL models approach as a initial specification for video compression standards. Development of CAL models require less time and the CAL model can be easily expanded or reused than Verilog/C languages for early estimation. This is due to data flow nature of CAL descriptions well suited for video compression algorithms. Furthermore, when it comes to video compression applications, this approach is domain specific can be easily extended to evolving video compression standards such as Scalable Video Codecs (SVC) and H.265 which is in its inception stage. This profiling framework can link all of these design space exploration steps by offering a centralized framework to the community and reducing the design efforts among communities of design engineers.

## 4. Conclusions and Future Work

The proposed design flow will provide an analytical and efficient design profiling methodology, capable of accurately profiling many different H.264 variants. This benefits various classes of designers including algorithm developers who are standardization committee members and hardware designers. In particular, designers use the profiling to derive extended CAL equations for existing different video compression algorithms since video compression algorithms share similar extended functions which have distance criteria and control strategies. This design flow can link all of these design space exploration steps by offering a centralized profiler to the community and reducing the design efforts among communities of design engineers.

Additionally, this design flow can integrate research and teaching activities including coursework development for design competition style classes. This will lead to revision of the current curriculum encouraging discussion of emerging design issues beyond conventional design metrics such as area, power and timing. This will provide students with an opportunity to understand the concept of system design tradeoffs by gaining hands-on experience using the profiler.

## References

[1] J.G. Tong and M.A.S. Khalid. A comparison of profiling tools for fpga-based embedded systems. In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pages 1687 –1690, 22-26 2007.

[2] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara. Specify-explore-refine (ser): From specification to implementation. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 586 –591, 8-13 2008.

[3] F. Angiolini, L. Benini, and A. Caprara. An efficient profile-based algorithm for scratchpad memory partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(11):1660 – 1676, nov. 2005.

[4] P. Grun, N. Dutt, and A. Nicolau. Access pattern based local memory customization for low power embedded systems. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 778 –784, 2001.

[5] `http://www.synopsys.com/Systems/`.

[6] Qubo Hu, E. Brockmeyer, M. Palkovic, P.G. Kjeldsberg, and F. Catthoor. Memory hierarchy usage estimation for global loop transformations. In *Norchip Conference, 2004. Proceedings*, pages 301 – 304, 8-9 2004.

[7] O. Ozturk, M. Kandemir, M.J. Irwin, and S. Tosun. Multi-level on-chip memory hierarchy design for embedded chip multiprocessors. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 8–14, 2006.

[8] P. Schaumont, D. Hwang, and I. Verbauwhede. Platform-based design for an embedded-fingerprint-authentication device. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(12):1929 – 1936, dec. 2005.

[9] `http://embedded.eecs.berkeley.edu/caltrop`.

[10] E. Ogura, Y. Ikenaga, Y. Iida, Y. Hosoya, M. Takashima, and K. Yamash. A cost effective motion estimation processor lsi using a simple and efficient algorithm. In *Consumer Electronics, 1995., Proceedings of International Conference on*, page 248, 7-9 1995.

# RSA Cryptography Acceleration for Embedded System

Rolando Duarte Chen Liu Xinwei Niu

*Computer Architecture and Microprocessor Engineering Lab (CAMEL)*

*Department of Electrical and Computer Engineering*

*Florida International University*

*Miami, FL 33174, U.S.A.*

*{rduar002, cliu, xniu001}@fiu.edu*

***Abstract*—Cryptography plays an important role for data security and integrity and is widely adopted, especially in embedded systems. On one hand, we want to reduce the computation overhead of cryptography algorithms; on the other hand, we also want to reduce the energy consumption associated with this computation overhead. In this paper, we explore techniques to improve the overall throughput and energy consumption of RSA (Rivest, Shamir and Adleman) public-key cryptography. Instead of implementing the entire algorithm into hardware format, we carefully implemented a custom coprocessor design to accelerate a single hotspot function of RSA algorithm on a Virtex5 FPGA platform. Then, we compare the effectiveness of the coprocessor design against the software implementation of RSA. The hardware accelerates the execution time by 10% thus minimizing the energy by 9%, achieving our goal.**

***Keywords-Coprocessor; cryptography; RSA; hardware accelerator.***

## I. INTRODUCTION

The Internet has evolved so fast that it not only provides information but also permits to communicate and make electronic transactions around the world. Hence, we want our personal data to be secured, reliable, and efficient over the Internet. Many cryptography algorithms have been implemented to prevent intruders from stealing information during an electronic transaction. They are widely used in applications such as ATM cards, computer password, e-mails and even within the world of electronic commerce. As secure communication bandwidth demands continue to grow, it requires faster cryptographic processing. This serves as the motivation for our hardware acceleration approach. Hardware acceleration hastens some specific operations, allowing the overall system, including the general purpose processor and the coprocessor, to execute concurrently in order to achieve performance improvement. The processor assigns specific function to the coprocessor while the processor executes its own instructions. For example, Irwansyah et al. [2] and Hodjat et al. [3] integrated the AES cryptography [4] as a complete system and interfaced it with the microprocessor. This technique is very efficient to accelerate as most as you can to finish executing the process as fast as possible. However, we need to take into consideration that adding more hardware to the system implies that it will consume more power. Clearly, there is a trade-off between performance improvement and power consumption. Also, what if the system requires some other hardware accelerator? Then it can be a critical part since this will incur even more power consumption.

Our design is based on FPGA platform, which allows us to customize our hardware implementation without going through the process of realizing the hardware into a physical chip. However, the resources on the FPGA are limited. If the system is complex enough with multiple accelerators needed and has occupied most of the FPGA resources, we might run out of space and probably will end up using several FPGAs to accommodate the customized hardware. Thus, we want a system that not only executes faster but also consumes less power and takes less space or resources in the FPGA platform. Nuan et al. [11], Hani et al. [12] and Zutter et al. [13] focused on speeding up the RSA [1] cryptography core by enhancing the modular exponentiation of square and multiplication. However, they implemented the whole RSA algorithm as a coprocessor. We followed a similar path, but instead of implementing the RSA cryptography core in its entirety, we selectively implemented only a single hardware accelerator targeted at a single hotspot function of the algorithm. We employed hardware in the form of a customized IP that accelerates the computation, so that we can observe a performance improvement when we execute the software code with the new integrated hardware.

The rest of the paper is organized as following: Section II will cover the system architecture of our design, which describes all the components that are used during the implementation of the hardware accelerator. This section also provides information why we chose to accelerate specific function as a custom IP and how it interfaces with the microprocessor. Section III presents the experiment results we obtained when we added the new peripheral to our system. Finally, the conclusion is drawn in Section IV.

## II. SYSTEM ARCHITECTURE

FPGA are widely used because its reconfigurability and reprogrammability can meet the different needs of the user. Thus, we chose FPGA because of its flexibility to add a hardware component into the device and its ease to reprogram the device. We could implement the same design using a simulator-based approach. However, we believe FPGA

platform could generate more accurate performance and energy consumption readings. We used the Virtex5 FPGA board [7] to interface our IP with Microblaze processor [8]. The overall system architecture is shown in Figure 1. Our system contains the following microprocessor, peripherals, and buses for intercommunication: Microblaze, BRAM, Local Memory Bus (LMB), Processor Local Bus (PLB), RS232, Timer, Interrupt and our customized accelerator. Microblaze is a soft-core microprocessor and it is of RISC architecture optimized for Xilinx FPGA boards. Microblaze is the only soft-core processor Virtex5 supports and it runs at 125 MHz. This processor is responsible for the execution of all instructions and communication among peripherals. BRAM is a Block RAM memory system with 64 KB memory space and the main purpose of this peripheral is to hold all instructions and data to be executed during the process. Microblaze access either instruction through ILMB or data through DLMB. These two buses are 32-bit wide. The LMBs are only connected to the Microblaze because it is the only component responsible for executing instruction and its data. The PLB provides communication between Microblaze and all the peripherals. If any peripheral needs to access another peripheral, PLB is the one accountable for this communication as well. RS232 is in charge of receiving and sending data to the user via HyperTerminal. Thus, we use RS232 to verify the results of RSA encryption/decryption between with acceleration and without acceleration approaches. We use a timer to gather information about how many clock cycles certain process takes. The interrupt peripheral is needed since we want our process to be interruptible because in an event that Microblaze needs to execute an instruction with higher priority, then any other peripheral can be interrupted. Finally, we have *Power_HW(2, n)* IP, which is our customized accelerator. It requires 12 slices and 133 LUTs. Thus adding the custom hardware, as shown in Figure 1, increased the usage of overall FPGA resource by 6% with respect to the hardware platform without acceleration. Microblaze will determine when our customized peripheral will be used and will be responsible for collecting the data provided by the customized hardware and sending the data to other peripherals connected through the same PLB bus.

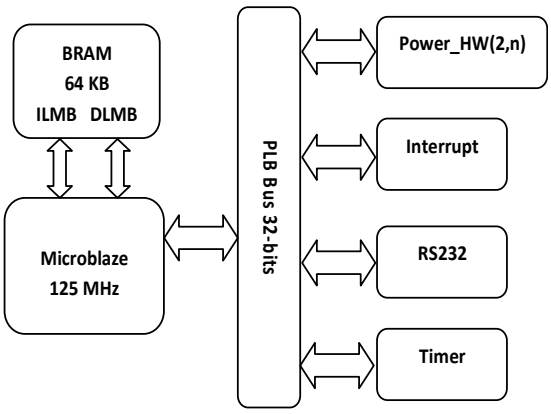



Figure 1. Power_HW IP Interfacing Microblaze

Since FPGA is reconfigurable, Microblaze can be specified to run up to 125 MHz, its maximum speed. Moreover, any peripheral connected through PLB has dual-port communication, meaning that each peripheral can send data to and receive data from the PLB. Then, if any other peripheral or another customized hardware should be connected, it would be listed at the right side of the bus like the other peripherals. It also should be kept in mind that the PLB runs at the same speed as Microblaze to keep reliability consistent. Hence, our customized IP will also run at 125 MHz since it is also connected to the PLB.

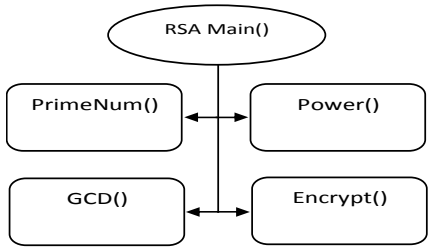



Figure 2. RSA Functions

The key point of our design is to identify which function executes most of the time and how long the entire process takes. The RSA software code in this specific design has four functions which are *PrimeNum()*, *GCD()*, *Encrypt()* and *Power()*, as shown in Figure 2. *PrimeNum()* provides a prime number every time it is called. RSA uses the multiplication of two prime numbers to decrypt and encrypt the data. The *GCD()* function determines the greatest common divisor since we know that two numbers are coprime if their greatest common divisor equals 1. This function is used in the process of generating the public key and private key. The *Encrypt()* method takes on the core mathematical operation of RSA algorithm, which is to calculate X to the power of Y modulo N. It performs either the encryption of the original data to convert it into a cipher data or the decryption of the cipher data to produce back the original data, depending on the parameter. *Power()* calculates 2 to the power of n, where n = 0, 1,…, 31. Based on the specific software implementation we use in this design, *Power()* actually is called by *Encrypt()* and its main functionality is to prepare the data into a specific format in order to calculate X to the power of Y modulo N. In [9], Chang et al. profiled the RSA algorithm using Intel® VTune™ Performance Analyzer [5] to gather information about which part of the software code takes most of the execution time, defined as hotspot function. They indicated *Power()* function as the hotspot function for RSA. Thus, we implemented a hardware accelerator that performs the same operation as software function *Power()* but the only difference is that our customized hardware will complete its task in fewer clock cycles. Consequently, it requires less execution time as we will see in the next section. Realizing all the RSA functions in hardware implies that the overall execution time will be faster. However, this will increase the usage of system resource, thus increasing the total power consumption. Our focus is to achieve the best speedup for the

overall system while minimizing the power and energy consumption, which naturally leads to the hotspot function acceleration.

## III. Experimental Results

The experiment consists of encrypting and decrypting 32-bit (4-byte) data. At first, the pure software code is executed without the existence of the customized hardware, and we observe that it takes approximately 39100 clock cycles to finish the process of encrypting and decrypting a 4-byte data, as shown in Table I. Since, Microblaze runs at 125 MHz, it takes about 0.31 ms to execute the entire process. Then, we followed the same process of encrypting/decrypting data, but each time the data size is incremented by multiple of ten to be consistent. Thus, the test set consists of data size from 4, 40, and all the way to 40000 bytes. As the data size increases, the number of clock cycles hence the execution time increases accordingly. They keep almost a constant relationship since the number of cycles for each data encryption/decryption is the same. The data-flow of the RSA software code is shown in Figure 3.

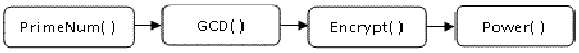



Figure 3. RSA Flow _ without Acceleration

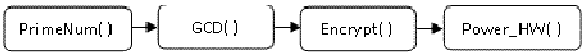



Figure 4. RSA Flow _ with Acceleration

Next, the Microblaze interfaces with our customized IP which is embedded into the FPGA. Partial of the RSA software code is now replaced by the corresponding hardware part. Here, the native software function *Power()* that computes 2 to the N*th* power is replaced by a coprocessor *Power_HW()*. The FPGA system is a memory-mapped system, thus each peripheral or embedded IP [6] is accessed by providing its corresponding memory address location. Hence, the *Power_HW()* is called upon by providing its memory location and the integer N. We execute the modified RSA software code to call the *Power_HW( )* instead, as shown in Figure 4. Now, the same data is loaded to *Power_HW()* as the one we used to call native software function *Power()*. The observation is that encrypting/decrypting 4 bytes of data takes 35000 clock cycles as shown in Table II. This leads to an execution time of 0.28 ms. The overhead of accessing the hardware accelerator through the PLB bus is 7 clock cycles plus 2 to 3 cycles to perform a load or store operation. The *Power_HW()* takes about 58 clock cycles to perform the operation of two to the power of N. RSA pure software function *Power()* takes about 118 clock cycle to finish the same execution and return the result. Hence, the custom IP performs 2 times faster than its pure software counterpart. If we compare the execution time of pure software approach with the hardware accelerator approach, we can examine that the overall speedup of our customize hardware is more than 10%, and as we increase the input size data, the speedup we achieves converges to a constant reading of 10.58%, as illustrated in Figure 5. We can apply Amdahl's Law [14] to verify the speedup we obtained while adding the custom hardware. Based on our observation, 22% of total execution time of RSA code is spent on *Power()* function, which is converted into hardware and this conversion acquires a speedup of 2, then we can see that the overall speedup = $\frac{1}{(1-p)+\,p/s}$, where p = 0.2 and s = 2. Then, the overall speedup is 1.1235, which also means the ideal speedup that can be obtained is 12.35%. We obtained an overall speedup of 10.58%. But we need to take into consideration that the timer is also connected to the PLB bus; thus it takes about 7 cycles to access the timer over the PLB bus. Therefore, this affects the overall execution time of the system and the overall speedup we achieve.

We utilized XPower Analyzer [10] to get the power consumption of the system. Xilinx claims that XPower Analyzer provides accurate power analysis after design implementation. The power consumption for the hardware system without the customized IP is 1. 2519 Watts and with the added IP is 1.2653 Watts, a 1% increase. Table I and Table II show the number of clock cycles and execution time of the two different implementations. Since we know the power consumed, we can calculate the energy for the RSA without customized IP approach and with the customized IP approach respectively. From these two tables, we can observe that our design executed the RSA algorithm effectively and the energy consumption is reduced after adding the embedded peripheral. Figure 6 shows the energy reduction of our hardware design over software. Moreover, the energy reduction converges to a constant reading of 9.62% with increased input data size. Hence, our hardware acceleration design not only gained speedup but also reduced the energy consumption. Figure 7 illustrates the normalized energy consumed per byte over the 4-byte input case, based on the data from Table II. We examined that the energy consumption per byte decreased as the data size increased.

TABLE I. RSA WITHOUT ACCELERATION

| # of Bytes | Clk Cycles (10^6) | Exec Time (ms) | Energy (mJoules) | uJoules / Byte |
|---|---|---|---|---|
| 4 | 0.0391 | 0.3127 | 0.3915 | 97.8795 |
| 40 | 0.3905 | 3.1241 | 3.9110 | 97.7756 |
| 400 | 3.8764 | 31.0109 | 38.8219 | 97.0547 |
| 4000 | 38.8182 | 310.5456 | 388.7658 | 97.1915 |
| 40000 | 388.2355 | 3105.8842 | 3888.1944 | 97.2049 |

TABLE II. RSA WITH ACCELERATION

| # of Bytes | Clk Cycles (10^6) | Exec Time (ms) | Energy (mJoules) | uJoules / Byte |
|---|---|---|---|---|
| 4 | 0.0350 | 0.2799 | 0.3541 | 88.5358 |
| 40 | 0.3494 | 2.7955 | 3.5372 | 88.4308 |
| 400 | 3.4656 | 27.7245 | 35.0809 | 87.7022 |
| 4000 | 34.7102 | 277.6816 | 351.3616 | 87.8404 |
| 40000 | 347.1555 | 2777.2442 | 3514.1582 | 87.8540 |

However, the energy consumption converges if data size is over 1 Kbytes. This is due to the factor that our system speedup also converges once the data size is above 1 Kbytes. Therefore, we can estimate how long it can take to encrypt and decrypt a larger data size of 400 Kbytes, 4 Mbytes, or even more. Basically we would expect the same speedup and energy reduction as shown in Figure 5 and Figure 6 respectively. If input data set is greater than BRAM capacity, it is required to put it into off-chip memory. In this case we employ a 256MB DDR memory. It takes 1074225 and 916194 clock cycles to encrypt/decrypt 4-byte data with and without acceleration respectively. If we compare with the BRAM case, BRAM-based design is more than 26 times faster than DDR-based design in both scenarios.

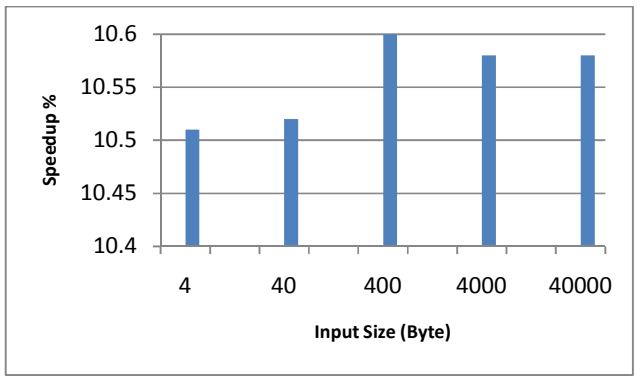



Figure 5. Speedup of with-Acceleration over without-Acceleration

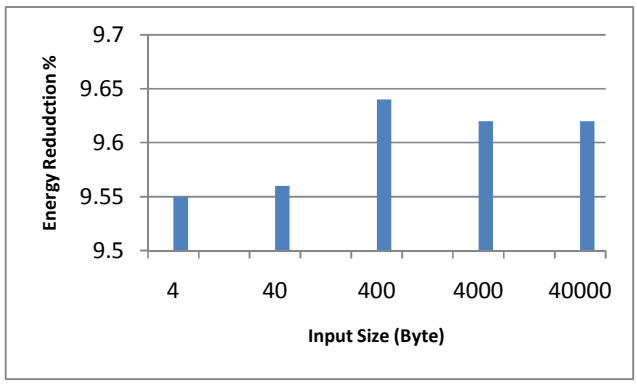



Figure 6. Energy Reduction of with-Acceleration over without-Acceleration

## IV. Conclusion

The implementation of an entire software algorithm into hardware is not always the best choice since we lose the reconfigurability and reprogrammability. We do not just want to accelerate our process but also want to reduce the energy consumption. In this paper, we explore the technique of identifying the hotspot function of a program and then realizing it as a hardware accelerator using RSA cryptography as an example design. The coprocessor helped the system to execute the specific function while the main processor executed the remaining of the code. We achieve an overall speedup of more than 10% and reduced its energy consumption by more than 9%. This technique can be implemented in other systems to explore ways of minimizing the hardware overhead and energy consumption as to maximizing its overall throughput. For future work, we are going to explore the hotspot functions for AES, Blowfish, MD5, 3DES and IDEA cryptography and implement them as customized hardware so that we can compare their execution time and energy reduction the same way we accomplished for the RSA cryptography.

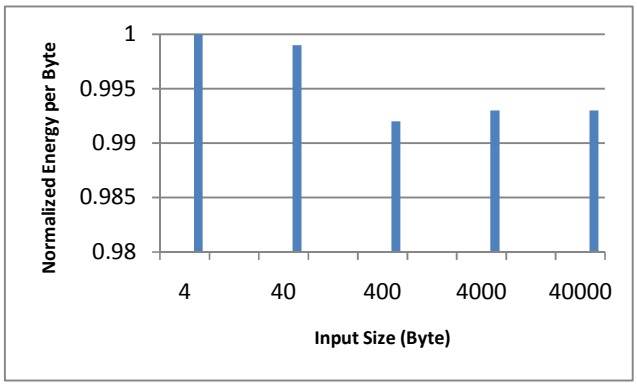



Figure 7. Normalized Energy per Byte Consumption with-Acceleration

## References


[1] Introduction of RSA for public-key cryptography. [Online]. Available: http://en.wikipedia.org/wiki/RSA

[2] Arif Irwansyah, Vishnu P. Nambiar, and Mohamed Khalil-Hani, "An AES Tightly Coupled Hardware Accelerator in an FPGA-based Embedded Processor Core," *Proc. ICCET'09*, vol. 02, p. 521-525, 2009.

[3] SAlireza Hodjat, and Ingrid Verbauwhede, "Interfacing a High Speed Crypto Accelerator to an Embedded CPU," *Asilomar SSC'04,*vol. 1, p. 488-492, Nov. 2004.

[4] Announcing the Advanced Encryption Standard (AES). [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[5] Intel® VTune™ Performance Analyzer. [Online]. Available: http://software.intel.com/en-us/intel-vtune/

[6] Paolo Giusto and Grant Martin, "Reliable Estimation of Execution Time of Embedded Software," *Proc. DATE'01,* p. 580-588, Mar. 2001.

[7] MicroBlaze Processor Reference Guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[8] Virtex-5 FPGA User Guide. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[9] Jed Chang, Chen Liu, Shaoshan Liu and Jean-Luc Gaudiot, "The Performance Analysis and Hardware Acceleration of Crypto-Computations for Enhanced Security," *PRDC'2010,* (accepted).

[10] Xilinx Xpower Analyzer [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm.

[11] Wen Nuan, Dai Zi Bin, and Shang Yong Fu, "FPGA Implementation of Alterable Parameters RSA Public-Key Cryptographic Coprocessor," ASIC'05, vol. 2, p. 769-773, 2005.

[12] Mohamed Khalil Hani, Tan Siang Lin, and Nasir Shaikh-Husin, "FPGA Implementation of RSA Public-Key Cryptographic Coprocessor," TENCON'00, vol.3, p. 6-11, 2000.

[13] Jan Zutter, Max Thalmaier, Martin Klein, and Karsten-Olaf Laux, "Acceleration of RSA Cryptographic Operations using FPGA Technology," DEXA'09, vol. 1, p. 20-25, 2009.

[14] The description and explanation of Amdahl's Law. [Online]. Available: http://en.wikipedia.org/wiki/Amdahl's_law