Second Workshop on

Computer Architecture and Operating System co-design (CAOS)

January 22, 2011

Herakleion, Crete, Greece

In conjunction with:

the 6th International Conference on

High-Performance and Embedded Architectures and Compilers

(HiPEAC)

Message from the Organizers

Welcome to the Second Workshop on Computer Architecture and Operating System co-design (CAOS), and thank you for helping us to make this event successful!

This meeting brings together researchers and engineers from academia and industry to share ideas and research directions in Computer Architecture and Operating System co-design and interaction. It is never easy to follow a successful edition, but this year we have three high-quality papers, spanning from single machine CMP systems to large data centers, and covering topics ranging from performance/power trade offs to memory walls and scalability for next generation systems. Power is surely this year's hot topic!

As last year, the second edition of CAOS presents another great keynote: "Reexamining TLB Design for Modern Chip Multiprocessors", from Prof. Margaret Martonosi (Princeton University). We hope to make great keynotes one of the CAOS's tradition.

This workshop is intended to be a forum for people working on both hardware and software to get together, exchange ideas, initiate collaborations, and design future systems. In fact, as multi-core and/or multi-threaded architectures monopolize the market from embedded systems to supercomputers, new problems have arisen in terms of scheduling, power, temperature, scalability, design complexity, efficiency, throughput, heterogeneity, and even device longevity. In order to minimize power consumption and cost, more and more cores per chip and hardware threads (contexts) per core share internal hardware resources, from the pipeline to the memory controller. Achieving high performance with these modern systems becomes increasingly difficult. Moreover, performance is no longer the only important metric: newer metrics such as security, power, total throughput, and Quality of Service are becoming first-order system design constraints.

It seems clear that neither hardware nor software alone can achieve the desired performance objectives and, at the same time, comply with the aforementioned constraints. The answer to these new challenges must come from hardware-software co-design. Computer Architectures (CA) and Operating Systems (OS) should interact through well-defined interfaces, exchange run-time information, monitor application progress and needs, and cooperatively manage resources.

We thank the Program Committee and the additional reviewers for their hard work and Omer Khan for his excellent work in putting together the proceedings for this edition.

Lamia Youseff (MIT)
Roberto Gioiosa (BSC)

Organizing Committee

Organization

Workshop Co-Chairs

Roberto Gioiosa	Barcelona Supercomputing Center	Spain	roberto.gioiosa@bsc.es
Lamia Youseff	MIT, CSAIL	USA	lyouseff@csail.mit.edu

Workshop Publicity & Publication Chair

Omer Khan	MIT, CSAIL	USA	okhan@csail.mit.edu

Program committee

Buyuktosunoglu, Alper	(IBM T.J. Watson, USA)
Cazorla, Francisco	(BSC, Spain)
Cesati, Marco	(U. of Rome Tor Vergata, Italy)
da Silva, Dilma	(IBM T.J. Watson, USA)
Etsion, Yoav	(Barcelona Supercomputing Center, Spain)
Hoffmann, Henry	(MIT, USA)
Holt, Jim	(Freescale, USA)
Kursun, Eren	(IBM T.J. Watson, USA)
McKee, Sally	(Chalmers University of Technology, Sweden)
Minnich, Ronald	(Sandia National Lab, USA)
Pakin, Scott	(LANL, USA)
Tsafrir, Dan	(Technion, Israel)
Villa, Oreste	(PNNL, USA)
Wisniewski, Robert	(IBM T.J. Watson, USA)

Web page

http://projects.csail.mit.edu/caos/

Advance Program

9.00-9.15	Opening
9.15-10.30	Keynote: **Reexamining TLB Design for Modern Chip Multiprocessors** *Prof. Margaret Martonosi (Princeton University)* The performance of Translation Lookaside Buffers is a very important factor in overall program performance, with TLB misses often representing 10% or more of program runtime. Despite this, their design issues remained largely unexamined as the transition from single-core to multi-core processors occurred. This talk will describe my group's recent body of work to characterize and optimize TLB behavior for Chip Multiprocessors. Considering both parallel workloads and also multiprogrammed workloads of sequential applications, we have proposed a range of techniques that can reduce TLB misses by more than 20% on average, with only modest hardware requirements. I will discuss hardware and software tradeoffs in the implementation of these ideas, as well as the methodological challenges of quantifying TLB performance effects. This talk will include many of our recent PACT '09, ASPLOS '10, and HPCA '11 results, and represents joint work with Abhishek Bhattacharjee and Dan Lustig.
10.30-11.00	Coffee break
11.00-11.30	**System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2)** Keun Sup Shim (MIT), Mieszko Lis (MIT), Myong Hyon Cho (MIT), Omer Khan (MIT), Srinivas Devadas (MIT)
11.30-12.00	**Power-Performance Adaptation in Intel Core i7** Vasileios Spiliopoulos (Uppsala University), Georgios Keramidas (Industrial Systems Institute), Stefanos Kaxiras (Uppsala University), Konstantinos Efstathiou (University of Patras)
12.00-12.30	**A Sleep-based Communication Mechanism to Save Processor Utilization in Distributed Streaming Systems** Shoaib Akram (FORTH-ICS), Angelos Bilas (FORTH-ICS)
12.30-14:00	Lunch

Keynote

Reexamining TLB Design for Modern Chip Multiprocessors

Prof. Margaret Martonosi (Princeton University)

Abstract:

The performance of Translation Lookaside Buffers is a very important factor in overall program performance, with TLB misses often representing 10% or more of program runtime. Despite this, their design issues remained largely unexamined as the transition from single-core to multi-core processors occurred. This talk will describe my group's recent body of work to characterize and optimize TLB behavior for Chip Multiprocessors. Considering both parallel workloads and also multiprogrammed workloads of sequential applications, we have proposed a range of techniques that can reduce TLB misses by more than 20% on average, with only modest hardware requirements. I will discuss hardware and software tradeoffs in the implementation of these ideas, as well as the methodological challenges of quantifying TLB performance effects. This talk will include many of our recent PACT '09, ASPLOS '10, and HPCA '11 results, and represents joint work with Abhishek Bhattacharjee and Dan Lustig.

Bio:

Margaret Martonosi is Professor of Computer Science at Princeton University, where she has been on the faculty since 1994. She also holds an affiliated faculty appointment in Princeton EE. Martonosi's research interests are in computer architecture and the hardware/software interface, with particular focus on power-efficient systems and mobile computing. In the field of processor architecture, she has done extensive work on power modeling and management and on memory hierarchy performance and energy. This has included the development of the Wattch power modeling tool, the first architecture level power modeling infrastructure for superscalar processors. In the field of mobile computing and sensor networks, Martonosi led the Princeton ZebraNet project, which included two real-world deployments of tracking collars on Zebras in Central Kenya. In addition to numerous publications, she has co-authored a technical reference book on Power-Aware Computing and six granted US patents. Martonosi is a fellow of both IEEE and ACM. In 2010, she received Princeton University's Graduate Mentoring Award.

System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2)

Keun Sup Shim Mieszko Lis Myong Hyon Cho Omer Khan Srinivas Devadas

Massachusetts Institute of Technology

Abstract. In this paper, we describe system-level optimizations for the Execution Migration Machine (EM^2), a novel shared-memory architecture to address the *memory wall* and *scalability* issues for large-scale multicores. In EM^2, data is never replicated and threads always migrate to the core where data is statically stored. This enables EM^2 not only to provide cache coherence without any complex protocols or expensive directories, but also to better utilize on-chip cache and thus experience much lower cache miss rate. However, it may incur significant execution migrations for shared data, which increases memory latency and network traffic, and thus, keeping migration rates low is a key under EM^2. We present systematic application optimization techniques to address this problem for EM^2 suitable for a compiler/OS implementation. Applying these optimizations manually to parallel benchmarks from the SPLASH-2 suite, we dramatically reduce the average migration rate for EM^2 by 53%, which directly improves parallel completion time by 34% on average. This allows EM^2 to perform competitively compared to a traditional cache-coherent architecture, on a conventional electrical network.

1 Introduction

The current trends in microprocessor design clearly indicate an era of multicores for the 2010s. As transistor density continues to grow exponentially, processor manufacturers are able to place a hundred cores (e.g., Tilera's Tile-Gx 100) on a chip with massive multicore chips on the horizon. Many industry pundits are predicting 1000 or more cores by the middle of this decade [5]. Will the current architectures (especially the memory sub-systems) scale to hundreds of cores, and will these systems be easy to program? Current memory architecture mechanisms do not scale to hundreds of cores because multicores are critically constrained by the *off-chip memory bandwidth wall* [5, 12]: the key constraint is the package pin density, which will not scale with transistor density [1]. Multicores to date have integrated larger caches on chip to reduce the number of off-chip memory accesses. Private caches, however, require cache coherence, and shared caches do not scale beyond a few cores [25].

Exposing the core-to-core communication to software for managing coherence and consistency between caches has limited applicability; therefore, hardware must provide some level of shared memory support to ease programming complexity. Snoop-based cache coherence does not scale beyond hundreds of cores. Directory-based hardware cache coherence requires complex states and protocols for efficiency; worse, directory-based protocols can contribute to the already costly delays of accessing off-chip memory because data replication and directory storage limits the efficient use of cache resources. S-NUCA [18] and its variants reduce off-chip memory access rates by unifying per-core caches into one large shared cache; accesses to memory cached in a remote core cross the interconnect and incur the associated round-trip latencies. The Execution Migration Machine (EM^2) [17], a general purpose shared memory architecture, instead migrates the computation's execution context to the core where the memory is (or is allowed to be) cached and continues execution there. Although moving execution context has a higher cost than moving data, EM^2 can outperform data migration architectures not only because memory accesses to a remote core require only one-way latencies instead of round-trip latencies, but also because successive memory accesses to the same remote cache—a frequent pattern under many modern applications with data locality—will result in one execution migration followed by a series of inexpensive local memory accesses.

The possible disadvantage of EM^2, however, is that since EM^2 restricts caching of each address to a single core, a large portion of data being shared within an application may cause significant migrations, which will increase both memory access latency and network load. In this paper, we extend the data alignment and replication techniques previously investigated in NUMA context (e.g., [27]) to the temporal dimension in order to improve migration rates and improve the overall performance under EM^2. Specifically:

1. We propose a limited-scope read-data replication optimization to reduce migration rates for an EM^2 architecture: when a shared address is read many times by several threads and seldom written, the proposed scheme allows temporary data copying to reduce the number of migrations. By taking advantage of the programmer's application-level knowledge, our replication can be applied to not only read-only pages but also read-write pages, and removes the process of page collapse (eliminating replicas on a write for read-write pages), which is a time-consuming requirement for page replication in NUMA architectures [27].
2. We show that applying the above mentioned optimizations to a baseline EM^2 architecture using a first-touch placement policy [21], lowers migration rates by 53% across the set of selected benchmarks. This improves the application performance, as measured by parallel completion time, by 34% on average. In contrast, our replication optimizations provide no benefits for cache-coherent systems because shared data are blindly replicated under cache coherence protocols.

2 The EM^2 architecture

Traditional hardware cache coherence multicore architectures bring *data* to the locus of the computation that is to be performed on it: when a memory instruction refers to an address that is not locally cached, the instruction stalls while the cache coherence protocol brings the data to the local cache and ensures that the address can be safely shared or exclusively owned. EM^2, on the other hand, brings the *computation* to the data: when a memory instruction requests an address not assigned to the current core, the execution context (architecture state and TLB entries) moves to the core that is *home* for that data. The physical address space in the system is divided among the cores, and each core is responsible for caching its region of the address space; thus, each address in the system is assigned to a unique core where it may be cached. (This assignment can, for example, be done by the OS on a first-touch basis, and is independent of the number of memory controllers). Since an address can be accessed in at most one location, ensuring properties that are difficult in traditional cache-coherent systems—such as sequential consistency and cache coherence—becomes simple. Under the same constraints of assigning each address to a unique core and not allowing local caching of remote data, moving the computation to data instead of bringing data to the computation has benefits because: (a) the execution migration is a one-way protocol whereas retrieving data requires round-trip latencies, and (b) for applications with data locality, successive memory accesses to the same remote cache will turn into *local* accesses under EM^2, whereas they would be repeated remote accesses under a remote-access design.

The cost of memory access within the EM^2 architecture is driven by the cost of memory accesses to the cache or DRAM, and the cost of migrations due to a *core miss*. A core miss is determined by computing the *home* core for a memory address. If the core that originated the memory access is the home, it is a *core hit*, otherwise, a *core miss*. The core miss cost incurred by the EM^2 architecture is dominated by transferring an execution context to the home core for any given address. Per-migration bandwidth requirements, although larger than those required by cache-coherent designs, are not prohibitive by on-chip standards: in a 32-bit x86 processor, the relevant architectural state amounts to about 1.5 Kbits including the TLB [24]. Although on-chip electrical networks today are not generally designed to carry that much data in parallel, on-chip communication scales well; further, the network can be optimized because all transfers have the same size and migrations are independent.

The per-memory-access cost can be expressed in terms of core hit and miss rates as

$$cost_{access} = rate_{core\_hit} \times cost_{memory} + rate_{core\_miss} \times (cost_{migration} + cost_{memory})$$
$$\text{where} \quad cost_{memory} = rate_{cache\_hit} \times cost_{cache} + rate_{cache\_miss} \times cost_{dram}.$$

While $cost_{dram}$ is relatively constrained, we can optimize performance by improving the other variables. Assignment of addresses to the cores determines the performance of an EM^2 design by influencing: (a) off-chip memory accesses required, and (b) pauses in execution due to migrations. On the one hand, spreading frequently used addresses evenly among the cores ensures that more addresses are cached in total, reducing cache miss rates and, consequently, off-chip memory access frequency; on the other hand, keeping addresses accessed by the same thread in the same core cache reduces migration rate and network traffic.

Prior work [17] shows that EM^2 improves $rate_{cache\_hit}$ when compared to a directory based cache-coherent configuration. However, reducing $cost_{migration}$ may require a high-bandwidth network, adding area as well as power to an already power constrained package. An alternative is to reduce $rate_{core\_miss}$, and that is the focus of this paper.

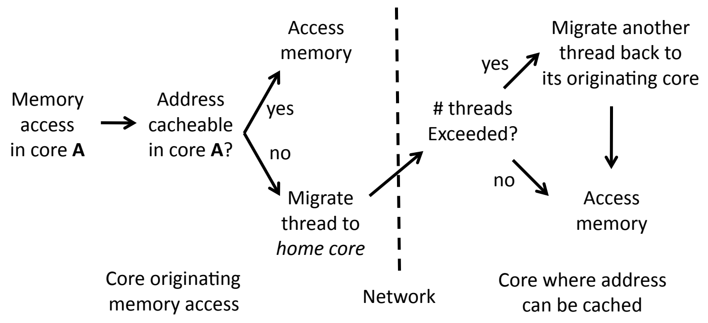

Memory access in core A
Address cacheable in core A?
yes
Access memory
no
Migrate thread to home core
Core originating memory access
Network
# threads Exceeded?
yes
Migrate another thread back to its originating core
no
Access memory
Core where address can be cached

Fig. 1. Memory accesses to addresses not assigned to the local core cause the execution context to be migrated to the core.

2.1 Data placement

Because under EM^2 each physical address resides in only one core and any attempt to access it will result in a migration to that core, the mapping of virtual addresses to physical addresses directly affects migration rates and cache utilization, and, consequently, memory access performance. The OS performs the mapping using the existing virtual memory mechanism: when a virtual address is first accessed and thus should be mapped to a physical page, it chooses where the relevant page should reside by mapping the virtual page to a physical address range assigned to a specific core.

In this paper, we use the ORIGINAL scheme, a variant of first-touch [21], where pages are mapped to the accessing thread's *originating* core on the first access, and remain there for the entire duration of the execution. The ORIGINAL scheme performs well because it aims to keep each thread on its originating core for as much of its running time as possible by taking advantage of data access locality, effectively reducing the migration rate while keeping the threads spread among cores.

2.2 Migration Framework

Figure 1 shows a slight variant of the migration framework of [17]. On a core miss (at say core *A*), the hardware initiates an execution migration transparent to the operating system. The execution context traverses the on-chip interconnect and, upon arrival at the home core (say core *B*), is loaded into the core *B* and the execution continues. In a single-threaded core, the thread running on the core *B* is evicted and migrated back to its originating core.

While this ensures that multiple threads are not mapped to the same core and requires no extra hardware resources to store multiple contexts, the context evicted from core *B* may well have to migrate back to core *B* at its next memory access. For this reason, we allow each core to hold multiple execution contexts, and resorting to evictions only when the number of hardware contexts running at the target core would exceed available resources. Results from [17] show that a 2-way multithreaded core microarchitecture (similar to [2]) provides sufficient performance by hiding the serialization effects of multiple threads contending for a core.

3 System-level Optimizations for EM^2

For non-trivially parallel applications, application optimization for a specific memory architecture is paramount in achieving the fastest possible performance, since it results in dramatic improvements in memory access latencies, a critical determinant of overall application performance. Although any shared-memory application can run on EM^2 without any modifications, applications resulting in significant migrations may suffer in performance since they will both increase the memory access latency and the network traffic.

In this section, we present OS- and application-level optimization techniques that significantly improve application performance by dramatically reducing migration rates for EM^2. We then show how these techniques apply in real application code by analyzing example benchmarks from the SPLASH-2 suite.

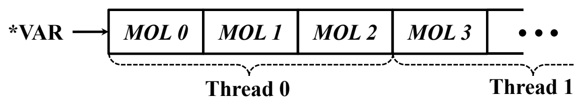
3.1 Optimization techniques

Per-thread heap memory allocation In most implementations, `malloc()` uses a shared heap to allocate memory to any requesting threads without regard to page boundaries: consecutive segments are assigned to different threads in the order in which the `malloc()` calls were invoked. Under EM^2 this can result in a kind of false sharing: private data used by separate threads are likely to end up on the same physical page and the threads will unnecessarily contend for that core.

When optimizing applications for EM^2, our goal is then to ensure that all thread-local data allocated using `malloc()` can be mapped to the thread that allocated them. With the ORIGINAL data placement scheme (described in Section 2.1), the address-to-core mapping occurs at a page granularity, and we can guarantee correct thread mapping by ensuring that `malloc()` calls in separate threads allocate memory from separate pages.

Operating system and library support for this optimization has two components: (a) ensure that `malloc()` and friends allocate data for separate threads in different pages, and (b) optionally allow the programmer to specify the thread to which the memory should belong. The first part is entirely transparent to the programmer, and consists of replacing the central dynamic memory management structure (say a free list) by a set of equivalent per-thread structures, and allocating data for each thread from its own pool. The second component exposes additional system details to the programmer, but works well in the common case where memory is first allocated in one thread and later different, disjoint regions are used in other threads (possibly spawned after memory has been allocated and initialized). This requires modifying the memory allocation (e.g., `malloc()`) and thread spawning (e.g., `pthread_create()`) library functions to take an additional parameter to identify the core where the memory should be mapped: for `malloc()`, this applies to the allocated memory, while in `pthread_create()` it applies to the newly created thread stack.

Restructuring data for private sharing In addition to overlapping sections of heap-allocated memory, data structures allocated contiguously by the programmer contain swathes of data private to different threads; for example, the WATER benchmark allocates an array of molecules processed separately by different threads:



—in this case, unless the molecule boundaries coincide with EM^2 page boundaries, false sharing will occur.

To improve EM^2 performance, the relevant data structure must be restructured (indeed, this is the same technique used to eradicate cache-line-level false sharing in the LU_CONTIGUOUS version of the LU benchmark). In most cases, this kind of transformation can only be done by the programmer, as the typical compiler would not, in general, be able to determine that different sections of the data structure are accessed by separate threads.

Read sharing and limited replication Some shared application data are written only once (or very few times) and read many times in multiple threads. In a cache-coherent architecture, this data will be replicated automatically in all user caches by the coherence protocol; under EM^2, however, each data element will stay in the core it was mapped to, and threads not running on that core will have to migrate there for access.

For example, several matrix transformation algorithms contain at their heart the pattern reflected by the following pseudocode:

```
barrier();
for (...) {
  ...     D1 = D2 + D3;     ...
}
barrier();
```

where `D1` "belongs" to the running thread but `D2` and `D3` are owned by other threads and stored on other cores; this induces a pattern where the thread must migrate to load `D2`, then migrate to load `D3`, and then again to write the sum to `D1`.

This observation suggests an optimization strategy for EM^2: during time periods when shared data is *read* many times by several threads and *not written*, make temporary local copies of the data and compute using the local copies:

```
barrier();
// copy D2 and D3 to local L2, L3
for (...) {
  ...     D1 = L2 + L3;    ...
}
barrier();
```

While a cache coherence protocol will do this blindly to all data regardless of how often it is read or written (and thus suffers high write-driven invalidation rates in benchmarks like RADIX), in EM^2 the programmer applies this technique judiciously using our profiling tool. The PIN-based profiler keeps track of the number of execution migrations for each code line, which tells the programmer which data are causing most migrations, and thus, better to be replicated. Since these local copies are guaranteed to be only read within the barriers by the programmer, there is no need to invalidate replicated data under our replication optimization.

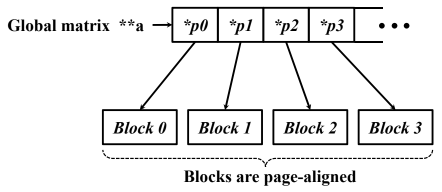
In our proof-of-concept SPLASH-2 benchmark refinements we applied this optimization by hand, and the copy process incurred many back-and-forth migrations. The number of these migrations can be significantly reduced by adding an architecture-level memory copy operation. Unlike string instructions present in some architectures (e.g., `movsb` and friends on x86) which are executed by the CPU, however, this operation would occur at the memory controller and would not involve any network traffic beyond the request itself and completion acknowledgement.

Architecturally, such an instruction would result in a message to the relevant DRAM controller requesting the transfer, and an acknowledgement-wait stall state if an instruction attempted to access the fresh copy of the data. If both memory ranges resided in the same memory controller, the copy would be internal to the controller and involve no traffic; if, on the other hand, the copied address ranges were mapped to two separate memory controllers, an efficient network-level block transfer would be used directly between the controllers. Finally, the memory controller would signal the requesting CPU core that the transfer has completed and accesses to the target memory range may proceed. In either case, the resulting network traffic would be significantly less than the many migrations required by "vanilla" EM^2 to complete the copy.

Because this architectural extension is not required for EM^2 functionality, however, the results we present here do *not* assume such an operation, and any data copy operations incur migrations as the copying threads bounce between the two relevant cores.

Specific benchmarks With these optimizations, we modified a set of SPLASH-2 benchmarks (FFT, LU, OCEAN, RADIX, RAYTRACE, and WATER) in order to reduce migration rate under the EM^2 architecture. Although we only describe our modifications for LU and WATER here, we have applied the same techniques for the rest of the benchmarks.

LU : In the original version optimized for cache coherence (LU_CONTIGUOUS), which we used as a starting point for optimization, the matrix to be operated on is divided into multiple blocks in such a way that all data points in a given block—which are operated on by the same thread—are allocated contiguously. Each block is also already page-aligned, as shown below:



Therefore no data restructuring is required to reduce false sharing.

During each computation phase, however, each thread repeatedly reads blocks owned by other threads, but writes only its own thread; e.g., in the LU source code snippet

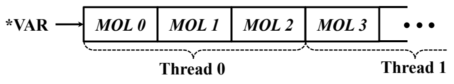
```
for (k=0; k<dimk; k++) {
  for (j=0; j<dimj; j++) {
    alpha = -b[k+j*strideb];
    for (i=0; i<dimi; i++)
```

```
        c[i+j*stridec] += alpha*a[i+k*stridea];
    }
}
```

since the other threads' blocks (a and b) are mapped to different cores than the current thread's own block (c), nearly every access triggers a migration.

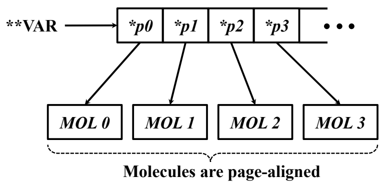
Since blocks a and b are read-only data within this function and the contents are not updated by other threads in the scope, we can apply the method of limited local replication as described in Section 3.1. In the modified version, a thread copies the necessary blocks—a and b in the example above—to local variables (which are also page-aligned to avoid false-sharing); the computation then only accesses local copies, eliminating migrations once the replication is done. We similarly replicate global read-only data such as the number of threads, matrix size, and the number of blocks per thread.

WATER : In the original code, the main data structure (VAR) is a 1D array of molecules to be simulated, and each thread is assigned a portion of this array to work on:



The problem with this data structure is that, as all molecules are allocated contiguously, molecules processed by different threads can share the same page and this false sharing can induce unnecessary migrations.

To address this, we modify the VAR data structure as follows:



By recasting VAR as an array of pointers, we can page-align all of the molecules, entirely eliminating false-sharing among them; this guarantees that, under EM^2, a thread never needs to migrate to access a molecule assigned to it.

In addition, WATER can also be optimized by locally replicating read-only data. For each molecule, the thread computes some intermolecular distances to other molecules, which requires read accesses to the molecules owned by other threads:

```
CSHIFT() {
  XL[0] = XMA-XMB;      XL[1] = XMA-XB[0];   XL[2] = XMA-XB[2];
  XL[3] = XA[0]-XMB;    XL[4] = XA[2]-XMB;   XL[5] = XA[0]-XB[0];
  XL[6] = XA[0]-XB[2]; XL[7] = XA[2]-XB[0];       ...
}
```

Here, XMB and XB are parts of molecules owned by other threads, while XMA, XA, and XL belong to the thread that calls this function. Since all threads are synchronized before and after this step, and the other threads' molecules are not updated, we can safely make a read-only copy in the local memory of the caller thread. Thus, after initially copying `XMB` and `XB` to thread-local data, the remainder of the computation induces no further migrations.

4 Methods

We use Pin [4] and Graphite [23] to model the EM^2 architecture. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [28] benchmark sets we use for evaluation, while Graphite models a tile-based core, memory subsystem, and network, as well as ensures functional correctness.

The settings used for the various system configuration parameters are summarized in Table 1. In experiments comparing EM^2 against cache coherence, the parameters for both were identical, except for (a) the memory directories which are not needed for EM^2 and were set to sizes recommended by Graphite on basis of the total cache capacity in the simulated system, and (b) the 2-way multithreaded cores which are not needed for cache coherent system.

Parameter	Settings
Number of cores	256, each with 2 threads, 1 issue-slot
L1/L2 data cache per core	16 KB/64 KB
Network	Mesh, 1 cycle per hop, 128 bit flits, XY Routing
Data placement scheme	ORIGINAL, VM page size 4 KB
Coherence protocol	Directory-based full-map MSI
Memory	30 GB/s bandwidth, 75 ns latency

Table 1. System configurations used

4.1 On-chip interconnect

Experiments were performed using Graphite's model of an electrical mesh network with XY routing. Each packet on the network is partitioned into fixed size flits, and we use the flit size of 128-bits for the electrical network. Since modern network-on-chip routers are pipelined [10], we argue that modeling a 1-cycle per hop router latency [20] is reasonable for the on-chip network; we account for the appropriate pipeline latencies associated with delivering a packet. In addition to the fixed per-hop latency, contention delays are modeled; the queuing delays at the router are estimated using a probabilistic model similar to the one proposed in [19].

4.2 Measurements

Our experiments used a set of SPLASH-2 benchmarks: FFT, LU, OCEAN, RADIX, RAYTRACE, and WATER. Each application was run to completion and used the recommended input set for the number of cores used, except as otherwise noted. For each simulation run, we tracked the total application completion time, the parallel work completion time, the percentage of memory accesses causing cache hierarchy misses, and the percentage of memory accesses causing migrations. While the total application completion time (wall clock time from application start to finish) and parallel work completion time (wall clock time from the time the second thread is spawned until the time all threads re-join into one) show the same general trends, we focused on the parallel work completion time as a more accurate metric of average performance in a realistic multicore system with many applications.

5 Evaluation

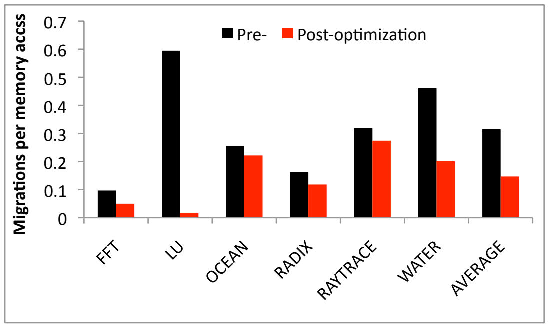


Fig. 2. EM^2 migration rates before and after the proposed optimizations. Because of better data distribution and judicious replication, migration rates drop significantly.

Figure 2 shows the effects of applying the optimizations described in Section 3. Distributing data on page boundaries to avoid false sharing, combined with judicious local replication of frequently used read-only data, combine to improve the average migration rate from 32% to 15% for the benchmarks we optimized for EM^2—a ca. $2\times$ improvement.

Although migration rate is not the only determinant of overall performance under EM^2, reducing the number of memory accesses that trigger migrations lowers the overall memory access time and significantly improves parallel completion times (Figure 3).

Figure 3 shows the overall parallel completion times for all benchmarks before and after our optimization. Before specifically optimizing the applications for EM^2, the EM^2 architecture was on the average outperformed by the cache coherence system; this is not very surprising, as most of these benchmarks have been specifically written with cache coherence systems in mind, and our choice of a network with 128 bit flit size. (Increasing network bandwidth beyond 128 bits benefits EM^2 much more than cache coherence [17].) After applying our optimizations, however, EM^2 on average performs competitively compared to the cache coherence system due to the significant drops in migration rates.

6 Related Work

Implicitly moving data to computation has been explored in great depth with many years of research on cache coherence protocols, and has become textbook material [13].

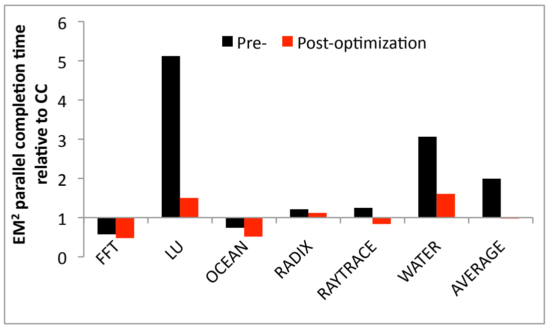


Fig. 3. In comparison to the directory-based, MSI cache-coherent system (CC), EM^2 performed 2× worse on average before optimizations. After optimizations, EM^2 performs competitively on average due to the significant drops in migration rates. The CC runs and the pre-optimization EM^2 runs used the original, cache-coherence-optimized SPLASH-2 benchmarks, while the post-optimization EM^2 runs used EM^2-optimized benchmark versions. EM^2 optimizations may worsen performance under CC and hence we used the original benchmarks for all CC simulations.

Meanwhile, page replication and migration have been extensively evaluated in the context of multiprocessor NUMA architectures. Verghese *et al* [27] propose OS supported dynamic page migration and replication to alleviate the problem of large remote access latencies in CC-NUMA architectures. In these NUMA systems, both interconnect and memory latencies were high and an OS-level approach provided sufficient performance; with today's fast on-chip interconnects, however, operating system interrupts are relatively much slower, and quick, low-overhead mechanisms are needed for good performance. Moreover, our replication optimization differs from prior NUMA research in that, using our profiling tool, we choose data to replicate by the access pattern that causes significant *migrations* and not by the number of *sharers*, because our focus is to reduce migration rates under EM^2. In addition, while the page replication in CC-NUMA requires the page collapse process to eliminate replicas on a write, the optimizations we present do not require this invalidation process since the replicated data are guaranteed to be only read in a limited scope by the programmer.

More recent research has explored data distribution and migration among on-chip NUCA caches [18] with traditional and hybrid cache coherence schemes. OS-level and OS-assisted software approaches [9, 12, 3, 6] leverage the operating system to map data to caches near where threads using it are scheduled (on the same core for private addresses and geographically close for shared data) and optionally replicate read-only pages. Other schemes add hardware support for page migration support [8, 26] or replication of recently used cache lines [29]. In general, only read-only pages are shared; in contrast, the optimizations we present here take advantage of the programmer's application-level knowledge to allow replication of read-write shared data during periods when it is not being written.

The idea of computation migration was originally considered in the context of distributed multiprocessor architectures [11], and has recently re-emerged in single-chip multicores for threads [22, 16] as well as thread segments [7]; compiler transformations for migration support have also been considered [15]. EM^2 [17] differs from these in that data sharing is completely abandoned (and therefore cache coherence protocols are not needed), and migration is required to provide memory coherence rather than employed to speed up access to cached data.

Finally, because of the complexity of coherence protocols and unscalable directory memory requirement, many recent many-core architectures (e.g., Intel's Single-chip Cloud Computer (SCC) [14]) rely on the message passing programming model instead of the shared memory model and give up on providing coherence support beyond software cache coherence. Message passing models, however, present the programmer with very low-level abstractions and, as they are relatively difficult to program, have historically been limited to specialized niche applications like scientific computing and telecommunications. In this paper, therefore, we have focused our optimization efforts on EM^2, a simple and scalable shared-memory architecture.

7 Conclusions and future work

In this manuscript, we have introduced a set of system-level optimizations to improve memory access latency for an EM^2 architecture: we chose a page-to-core mapping strategy, outlined several optimization techniques, and evaluated their effects on benchmarks from the SPLASH-2 suite. Our results show that these optimizations significantly reduce migration rates, which effectively improves the performance, enabling an EM^2 architecture to perform competitively compared to a traditional cache coherent system.

Our future research directions include automating the techniques presented here via a combination of low-overhead compiler, operating system, and/or architecture implementations; we believe a combination of the three will be critical in overcoming the limitations of previously explored migration/replication techniques. Furthermore, we will also consider how these optimizations can be applied to other application domains with different memory access patterns (e.g., streaming applications).

References

1. Assembly and packaging. *International Technology Roadmap for Semiconductors*, 2007.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. smith. The Tera Computer System, 1990.
3. M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *HPCA*, pages 250–261, 2009.
4. M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 43:34–41, 2010.
5. S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.
6. S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *HotOS*, 2009.
7. K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS*, pages 283–292, 2006.
8. M. Chaudhuri. PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, pages 227–238, 2009.

9. S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-Level page allocation. In *MICRO*, pages 455–468, 2006.
10. W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.
11. H. Garcia-Molina, R. Lipton, and J. Valdes. A massive memory machine. *IEEE Trans. Comput.*, C-33:391–399, 1984.
12. N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, pages 184–195, 2009.
13. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, September 2006.
14. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108 –109, Feb. 2010.
15. W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *PPOPP*, pages 239–248, 1993.
16. M. Kandemir, F. Li, M. Irwin, and S. W. Son. A novel migration-based NUCA design for chip multiprocessors. In *SC*, pages 1–12, 2008.
17. O. Khan, M. Lis, and S. Devadas. EM^2: A Scalable Shared-Memory Multicore Architecture. *MIT-CSAIL-TR-2010-030*, 2010.
18. C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.
19. T. Konstantakopulos, J. Eastep, J. Psota, and A. Agarwal. Energy scalability of on-chip interconnection networks in multicore architectures. *MIT-CSAIL-TR-2008-066*, 2008.
20. A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha. A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator. In *in 65nm CMOS, ICCD*, 2007.
21. M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *IPPS*, 1995.
22. P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, pages 186–195, 2004.
23. J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.
24. K. K. Rangan, G. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ISCA*, pages 302–313, 2009.
25. S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora. A 45nm 8-core enterprise Xeon® processor. In *A-SSCC*, pages 9–12, 2009.
26. K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38(1):219–230, 2010.
27. B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. *SIGPLAN Not.*, 31(9):279–289, 1996.
28. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.
29. M. Zhang and K. Asanović. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, pages 336–345, 2005.

Power-Performance Adaptation in Intel Core i7

Vasileios Spiliopoulos[1], Georgios Keramidas[2], Stefanos Kaxiras[1], and Konstantinos Efstathiou[3]

[1]Uppsala University, Sweden
[2]Industrial Systems Institute, Greece
[3]University of Patras, Greece

Abstract. In this paper, we describe our experiences in building a framework for power/performance run-time management for the Intel core family. Our underlying methodology (in contrast to previous work which relied on empirical models) is based on a simple processor performance model in which frequency scaling is expressed as a change (in cycles) of the main memory latency. Based on this model and utilizing performance monitoring hardware, the proposed model is shown to be powerful enough to i) describe and explain how Intel processors are affected by frequency scaling with respect to workload behavior, ii) predict with reasonable accuracy the effect of frequency scaling (in terms of performance loss), and iii) predict the energy consumed by the core under different *V/f* combinations by directly measuring from the off-chip voltage regulator the power consumed by the core. Our long-term plans include integrating in the proposed framework various power-aware OS/application-driven DVFS policies. As a first step towards this direction, we show our experimental methodology to justify the power/performance measurements and verify the correctness of our framework in which any target DVFS policy can be embedded as kernel module.

1 Introduction

The power-aware architecture landscape has been dominated by techniques based on supply voltage and clock frequency scaling. Dynamic Voltage and Frequency Scaling (DVFS) offers great opportunities to dramatically reduce energy/power consumption by adjusting both voltage and frequency levels of a system according to the changing characteristics of its workloads. The great potential of DVFS in energy/power savings has been widely studied in a variety of research communities (from circuit to system designers) and has been extensively used in commercial systems as well. Intel XScale, AMD Mobile K6, and Intel Pentium M are typical low-power processors that feature DVFS management capabilities. Example processors from the high-performance area are the AMD Opteron quad-core and the Intel core i7 processor.

In general, the heart of DVFS techniques is the exploitation of the system *slack* or "*idleness*." Their objective is to take advantage of slack so that performance is affected little by frequency scaling while at the same time a cubic benefit in power consumption —with the help of voltage scaling— is achieved [8]. Slack can appear at different levels and various approaches have been proposed for each level. According to [8], DVFS decisions can be taken at: i) the system level based on system slack, ii) the program level based on instruction slack, and iii) the hardware level based on hardware slack. More details about the criteria used to devise this categorization can be found in [8]. In this work, we are concerned with the instruction slack due to the long latency memory operations (off-chip memory accesses).

In our previous work [9], we developed two simple analytical models that are able to drive run-time DFVS decisions for aggressive superscalar OoO processors. These models work at the microarchitectural level and their target is to exploit the slack due to the long-latency, off-chip memory operations. *The realization that inspired the development of these models was that core frequency is nothing more than changing the memory latency in cycles*. This conceptual view of frequency scaling significantly simplifies the DVFS management decisions even for highly-aggressive, highly-pipelined, dynamic processors (e.g. the Intel core i7 [3]).

Previous approaches [4][5][6][11][12] in the area rely on empirical models requiring large profiling, training and trial-and-error steps or significant compiler assistance [10]. For example, the model proposed in [6] is prohibitively costly for run-time power estimation and optimization. It requires four complete program executions with different counter configurations, in order to collect the necessary information. In contrast, our models require minimal input and calculations [9]. The reason for this is that our models are able to acknowledge and isolate the processor events that directly correlate to DVFS processor behavior. Consider for example, the penalty of a branch misprediction. This penalty (measured in cycles) will remain intact no matter what the frequency is, because a branch misprediction involves only in-core operations. The penalty of in-core operations is always the same (measured in cycles) during frequency scaling [9].

The simple nature (minimal input and calculations) and the high accuracy of our models [9], inspired us to move one step forward. While our previous work was conducted in a cycle accurate simulator (equipped with the appropriate power models), in this work we provide our experiences and application results in applying those models in a real-life processor: the i7 Intel Nehalem core [3]. Testing research ideas in real processors was motivated by the integration of a rich set of performance monitoring counters which resides in almost all modern processors. It is well known that cycle-accurate simulations are very time consuming and their accuracy is a subject of considerable debate. Consider for example thermal studies where it takes a long time for processors to reach equilibrium thermal operation points. Live measurements allow a complete view of operating system and I/O effects and many other aspects of “real-world” behavior, often omitted in simulation. However, measuring live, running complex systems (i7 920 is a quad core SMT CMP) and relating measured results to overall system hardware and software behavior is not so straightforward as in a simulator, because many details are omitted from the computer vendors. *As a result a systematic approach is required to reverse-engineer the hardware details of the target processors.*

In this work, we provide a framework for power and performance run-time management for the Intel processors. Our framework can be formed as a basis for future power-aware research. As a first step towards this direction, we provide our experimental methodology to justify the power/performance measurement and verify the correctness of our framework in which any target DVFS policy can be embedded in the OS as kernel module. Some of the points, we try to shed light on, are: i) How much power (static and dynamic) is consumed by the core and the uncore areas of the processors? ii) How Intel processors are affected by frequency scaling with respect to the behavior of the applications? iii) Is the performance monitoring hardware appropriate for power-oriented optimizations? iv) How much clock-gated is the i7 core?

To quantify the robustness of the proposed framework, in this work we investigate the following scenario: we run an application in a specific V/f point collecting performance and power measurements (see Section 4 and 5). Based on those measures, we evaluate our methodology in predicting the performance and energy characteristics of the application in any given frequency/voltage combination (specified for example by the user or the OS).

Structure of the paper—Section 2 provides an overview of our previous work [9]. Section 3 discusses power-related details of the i7 core and presents our measurement methodology. Section 4 analyses the effect of frequency scaling in i7 core with respect to power "behavior" of the applications. Section 5 provides our experiences in predicting the energy under different *V/f* points. Section 6 outlines the presented methodology and discusses our future work. Section 7 concludes the paper.

2 Interval-based Analytical Models for DVFS Management

In our previous work [9], we showed that a successful way to model DVFS management in an OoO, dynamic processor is to *account only for the stall cycles introduced in the machine due to off-chip non-overlapping misses (Last-Level Cache or LLC misses)*. The idea is that only these misses directly correspond to the stall cycles that are affected by the processor's frequency. Based on this, we introduced a model, called miss-based model, which takes as input the number of stalls introduced in the machine due to LLC misses and outputs the execution time —and the energy— under different frequencies with less than 1% (avg.) error. We also introduced a more simplified model, called stall-based model, which is not able to distinguish pipelining of the LLC misses (i.e., it accounts for stalls for both isolated and overlapping misses). The stall-based model still yields acceptable results (5% on avg.). A deeper examination of this model shows that the extra error is introduced because the model disregards useful work performed by the processor when a LLC miss occurs (i.e. from the occurrence of the miss until no more instructions can enter the execution window or all the instructions in the execution window are dependent on the pending miss). However, the latter model offers a great potential: it can be used in real-life processors (e.g. the i7 core), in contrast to the miss-based model (the current hardware monitoring events are not able to distinguish between overlapping and isolated misses [2]).

Our modelling methodology is inspired by the interval-based performance model [1][7]. Intervals are marked by miss-events that upset the "steady state" execution of the program. A *miss-interval* starts with a miss-event (LLC misses in our case) and lasts until the IPC reaches again a steady state (a period related to the memory latency). Periods between miss-intervals are steady–state intervals. *The realization that drives this work is that core frequency scaling in these models is nothing more than changing the memory latency in cycles.* Figure 1.a shows the different areas of a LLC miss interval. The stall-based model takes as input the cycles which correspond to the *full stall+IQ Drain* areas and assumes that this quantity is equal to memory latency measured in cycles — it disregards the ROB fill area. Note that this area, measured in cycles, remains intact in all frequencies. The error of the stall-based model can be seen in the following formula:

$$Stall_{cycles} = Mem_{lat} - ROB_{fill} \approx Mem_{lat}$$

The sum of stalls in overlapping misses (Figure 1.b) is also approximated to memory latency:

$$ST1 + ST2 = y + Mem_{lat} - ROB_{fill} - x \approx Mem_{lat}$$

The conclusion is that *the sum of stall cycles is proportional to memory latency and thus proportional to frequency scaling.* On the other hand, non-stall cycles remain

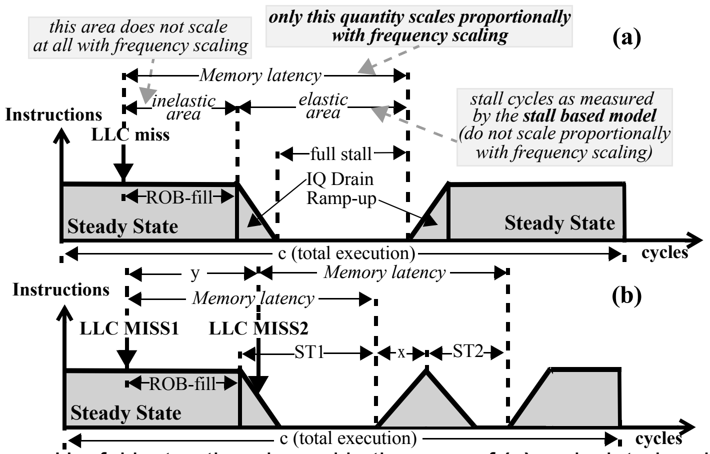


Figure 1 Useful instructions issued in the case of **(a)** an isolated and **(b)** overlapping LLC load misses.

intact. With these observations in mind, extracting the formula for predicting execution time is straightforward (more details can be found in [9]).

The miss-based model acknowledges that the miss interval equals memory latency and thus scales proportionally to frequency. Furthermore, it is able to recognize that *only the miss interval of the first miss in a cluster of misses scales with frequency, while the miss intervals of overlapping misses remain intact with frequency scaling.* Another way to express this is that if a miss occurs *y* cycles (Figure 1.b) after the initial miss it will also be serviced *y* cycles after the first miss is serviced *so the extra stall cycles introduced by the overlapping miss do not change with frequency.* When the miss that headed a cluster of overlapping misses is serviced, the next miss in line starts a new cluster *even if it overlaps with an outstanding miss from the previous cluster*. The methodology followed by the miss-based model is similar to the stall-based model, but instead of stall cycles, the quantity that scales proportionally to the frequency is the number of clusters of misses multiplied by the memory latency. More information can be found in [9].

3 How and What to Measure in Intel Core i7

3.1 Intel Core i7: main features

Intel core i7 is a quad-core CMP. Each core supports hyperthreading execution. i7 Core family is enhanced with a special power-aware characteristic, called *Speedstep technology* [3] which allows run-time voltage and frequency scaling between 9 different steps, from 1.6 to 2.66GHz (i7 920). i7 supports also various idle states, called C-states, in which it is possible to completely deactivate the clock and cut-off the power supply for a combination of cores to reduce static and dynamic power consumption.

3.2 Performance Counters in i7

i7 core offers a wide range of hardware monitoring event counters. Table 1 shows the performance counters used in this work. One of our main problems in this work was

Performance Counter	Description
UOPS_EXECUTED.CORE_STALL_CYCLES	cycles no insts are executed in the processor
L2_RQSTS.LD_MISS	load requests that missed L2 cache
LLC_MISSES	last level cache misses
BRANCH_MISSES.RETIRED	mispredicted branches
UOPS_EXECUTED.PORT015	micro-ops executed in ports 0, 1 or 5
UOPS_EXECUTED.PORT234	micro-ops executed in ports 2, 3 or 4
UNHALTED_CORE_CYCLES	cycles core not halted

Table 1: The Intel i7 core hardware events selected for this work.

Number of Active Cores	2.66 GHz (Nominal Freq.)	1.6 GHz (Minimum Freq.)
4 cores	15.8W	7.6W
2 cores	10.4W	2.1W
1 core	2.6W	1W

Table 2: Power consumed by the i7 cores in the idle state.

that there is no performance counter in i7 core to account for the stalls introduced in the machine due to LLC non-overlapping misses (Section 2). In other words, there are no specific performance counters to measure the *Memory-Level Parallelism* of the LLC misses (also pointed out in [2]). As a result, we could not use our highly accurate miss-based model. Therefore, we used an approximation of the stall-based model in order to predict the performance under different *f* points using the counter event mentioned in line 1 in Table 1. Finally, all the other performance counters listed in Table 1 are used only for cross-checking our results (not to predict the effect of frequency scaling) and to gain a better understanding of the behavior of the running applications.

3.3 Measuring Power/Energy in i7

i7 core comprises of two main voltage islands: core (exec. and fetch units, OoO and paging logic, L1/L2 caches and branch prediction) and the uncore (L3 caches, memory controller and QPI). In order to isolate the core and the uncore power, we compute core power dissipation by directly measuring voltage and current from the off-chip voltage regulator (ADP4000) residing in the motherboard by identifying two pins of interest: the pin that supplies the voltage to the core and the pin monitoring the total output current of the regulator. By hacking the motherboard (connecting wires to these pins) we were able to measure power using an oscilloscope while the processor was under normal operation. We use a sampling period of 10ms (our target is to provide OS-level optimizations so finer granularities will not provide useful results). The power measurements can be easily fed to the kernel OS using DLP-IO8, a USB analog–to–digital converter. Our future work includes utilizing this information (in the kernel level) to drive application/OS-driven DVFS policies.

3.4 Static Power in i7

When the processor is in the idle state, it consumes only static power since the clock is cut-off (the off-chip voltage regulator still provides voltage to the core). To get a full picture of how much power is consumed in the idle state, we deactivate different number of cores from the BIOS. The assessed idle power under different frequencies is gathered by our kernel module. The collected power numbers can be then used for predicting the processor power at run-time. Table 2 shows idle power for different number of cores under maximum and minimum frequency.

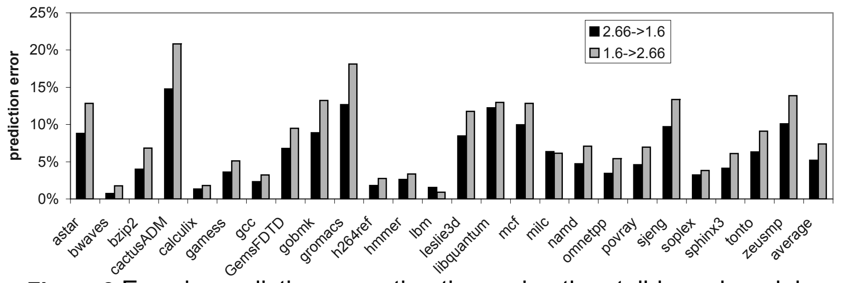


Figure 2 Error in predicting execution time using the stall-based model.

3.5 Applications and OS

We run our experiments on an Ubuntu Linux 9.10 system with the 2.6.31-22 kernel. The kernel is patched to enable our techniques to run as kernel modules. We use the entire SPEC2006 suite with all the ref. inputs. We compiled the benchmarks with gcc 4.3 as 64-bits binaries and -O3 optimization. We use full benchmark runs to get a complete view of the benchmark behavior (the benchmarks run for several minutes in our machine). Finally, the measurement of the performance counters runs as a kernel module, enabling counting in the OS. This way, no changes to the target applications are required and the timing overhead during execution remains minimal (less than 1%).

4 Predicting Performance for Different *f* Points

Figure 2 shows the absolute error of predicting the execution time using our stall–based model in i7 core for a large frequency step: running the program in the nominal frequency and predicting the execution time in the minimum frequency (black bars) — the grey bars represent the reverse scenario. Due to space restrictions, for the benchmarks with multiple inputs, we present in Figure 2 the average error over all inputs. To further analyze the results, we classify the benchmarks into three categories: *CPU–bound*, *memory–bound*, and *intermediate* or *mixed* category. This categorization is performed as follows: when the frequency is scaled from 2.66 to 1.6 GHz, a purely CPU-bound program will suffer an increase in its execution time of 66.67%, but due to memory accesses this penalty will be smaller. Based on this, a program with performance penalty of more than 55% (when scaling the frequency from max to min) is CPU-bound, a program with penalty less than 35% is memory bound while the rest of the benchmarks fall into the intermediate category.

In general, the more memory–bound a program is, the more the increase in the prediction error. This is an inherent property of the stall–based model, since this model ignores the ROB-fill effect. In the rest of this section we explain errors on a per-benchmark basis. To be able to delve into details about the power "behavior" of each benchmark, we also gather information for all the events listed in Table 1.

CPU bound—This category contains the following benchmarks/inputs: *astar_2*, *bwaves, bzip2* (6 inputs), *cactusADM*, *calculix*, *gamess* (3 inputs), *gobmk* (5 inputs), *gromacs, h264ref* (3 inputs), *hmmer* (2 inputs), *namd, povray, sjeng, sphinx3, tonto and zeusmp*. The errors in predicting the execution time in this category for 17 benchmarks are below 5%. Clear exceptions are *cactusADM* (14.8% error), *gromacs* (12.7%), and *astar_2* (10.9%). To understand and explain these errors, Table 3 shows

CATEGORY	BENCHMARK	TIME PENALTY (%)[A]	STALLS	L2 MISSES	L3 MISSES	BRANCH MISPRED	IPC[B]	**ERROR (%) FROM 2.66 TO 1.6 GHZ**
CPU BOUND	CACTUSADM	63.4	356	2.7	2	0.02	2	**14.8**
	GROMACS	65.8	287	2.2	0.01	4.96	1.97	**12.7**
	ASTAR_2	67.1	234	6.36	0.09	23.66	2.28	**10.9**
	ZEUSMP	57.4	348	3.22	2.5	17.84	1.83	**10.1**
	SJENG	63.3	260	0.96	0.5	30.7	2.26	**9.7**
	BZIP2_3	60.8	127	11.4	0.02	28.4	2.33	**2**
	H264REF_1	66.1	37	1.35	0.05	13.8	2.49	**1.4**
	GAMESS_2	65.3	46	0.16	0	27.2	2.5	**1.3**
	CALCULIX	65.8	39	1.12	0.1	41	2.54	**1.3**
	BWAVES	58.4	134	1.04	1	12.47	2.39	**0.7**
MIXED	MCF	44.8	518	25	10.4	9.1	1.47	**10**
	LESLIE3D	52.9	378	4.1	3.6	0.36	1.9	**8.5**
	GEMSFDTD	52.4	353	7.6	4.3	0.3	1.67	**6.8**
	ASTAR_1	52.1	357	10.8	3.9	14.4	2.13	**6.7**
	GCC_2	54.6	304	6.72	1.67	17.1	2.13	**6**
	LBM	48	231	1.87	2.88	6.75	1.78	**-1.5**
	GCC_5	37.4	394	4.35	5.36	7.15	2.11	**-1.3**
	GCC_3	35.9	421	3.34	4.45	7.26	2.15	**-1.2**
	GCC_4	43.3	336	4.52	5	10.15	2.14	**-0.2**
	GCC_6	38.4	405	4.21	5.25	6.94	2.14	**-0.1**
MEMORY BOUND	LIBQUANTUM	17.4	483	3.27	5.58	4.97	1.6	**-12.2**
	MILC	22.5	519	10.9	12.9	3.19	1.99	**-6.4**
	GCC_8	28.5	472	3.95	4.89	5.96	2.18	**-4.1**
	OMNETPP	33	565	14.7	7.35	6.49	2.04	**3.4**
	GCC_7	32.2	480	3.18	5.24	7.11	2.1	**-1.3**

Table 3: Performance counter events (per 1K cycles) and prediction error for three benchmark categories: CPU bound, mixed, and memory bound category.

a. Performance penalty when the frequency is scaled from 2.66 GHz to 1.6 GHz.
b. Instructions per non-stall cycles (indicates Instruction Level Parallelism).

the results for the full list of the performance counters for representative cases. By cross comparing the results, we are able to explain the behavior of each benchmark.

The error in *cactusADM* is due to the increased number of stalls generated by L2 and LLC misses and the low IPC. Although only 2 LLC misses per 1k cycles occur, the penalty for a *fmax* to *fmin* transition is 63.4% which means that these L3 misses overlap either with each other or with L2 misses, thus few of the stalls counted are *strictly* due to LLC misses. *gromacs* shows a similar behavior. The increased number of stalls are due to L2 misses, branch mispredictions and low IPC. On the other hand, *astar_2* has a quite high IPC which “hides” the stalls due to the large number of L2 misses thus the prediction error is smaller (10.9%). *zeusmp* lies on the border between CPU-bound and mixed categories, so the 2.5 LLC misses per 1k cycles introduce some stall cycles but L2 misses, branch mispredictions and low IPC also introduce stall cycles resulting in an error of 10.1%. Finally, *sjeng* has few misses but many branch mispredictions and as a result a 9.7% error. In general, benchmarks with few miss events and high IPC exhibit smaller errors. For example, *h264ref1* and *calculix* have about 1 L2 miss per 1k cycles and 13.8 and 41 branch mispredictions respectively, but due to high ILP (indicated by high IPC) few stalls are introduced in the machine resulting in low errors. Similar behavior is reported by *gamess_2*. *bwaves* approaches the mixed category, but the extra stalls introduced by L2 misses and branch mispredictions boost the model to compensate for its inherent inability to account for the ROB-fill area (this phenomenon will be further explained in the next categories).

Mixed category—This category includes: *astar_1*, *gcc* (6 inputs), *gemsFDTD*, *lbm*, *leslie3d*, *mcf* and *soplex* (2 inputs). As we move towards more memory-bound programs, we observe that the prediction error becomes smaller. This is because now the stall cycles introduced by L3 misses become a larger part of the total stall cycles and what we measure is closer to what we should measure according to the stall-based model. *mcf* yields the largest prediction error in this category (10%) due to the large amount of L2 misses (25), the branch mispredictions (9.1) and the strong dependencies between instructions illustrated by the low IPC (1.47). Although there are many L3 misses (10.4), *mcf*'s penalty is only 44.8%, which means that most of them are not performance critical (overlapping misses). *leslie3d* and *gemsFDTD* are more CPU-bound compared to *mcf* and the prediction error is smaller due to the reduced amount of miss events (4.1 and 7.6 L2 misses and 0.36 and 0.3 branch mispredictions respectively). *astar_1* has about the same penalty with *leslie3d* and *gemsFDTD* and more miss events (10.8 L2 misses and 14.4 branch mispredictions), but it also has larger IPC (2.13) which means that the processor is able to keep executing instructions when a miss event occurs and thus the majority of stalls measured are due to L3 misses. Similar to *astar_1* is *gcc_2* which has a slightly smaller prediction error due to the reduced amount of L2 misses.

Until now we explained how miss events other than L3 misses pollute our measurements. As the program becomes more memory-bound, L3 misses become the governing factor in stalls and another source of inaccuracy arises: neglecting the ROB-fill area. Although this is an inherent problem of the stall-based model, the way we measure stalls improves accuracy because the extra stalls measured due to other miss events compensate for the non-measured ROB-fill area. The negative error indicates that the stall cycles are underestimated. So in *lbm*, L2 misses, branch mispredictions, and low IPC produce extra stalls which reduce the error. The error would be even smaller if more stalls were measured. This is the case for *gcc_3*, *gcc_4*, *gcc_5*, *gcc_6*. The stalls introduced by L2 misses and branch mispredictions result in small prediction error, less than 1.5%.

Memory bound category—This category includes 5 benchmarks. The largest prediction error is observed in *libquantum* (12.2%). The low increase in execution time (17.4) between the maximum and the minimum frequency indicates that *libquantum* is heavily memory bound, although L3 misses are not that many. This means that all or most of them are performance critical (isolated). *libquantum* has a few other miss events which reduce the error (negative error means that stalls are underestimated and extra stalls reduce prediction error). *milc* is slightly more CPU-bound compared to *libquantum* but also has more L2 misses, so the extra stalls reduce prediction error to 6.4%. *gcc_8* is more CPU-bound and the prediction error is improved. *omnetpp* and *gcc_7* have about the same penalty, but differ in the sign of prediction error. *omnetpp* has many L2 misses which results in overestimating the stalls and thus positive errors. On the other hand, *gcc_7* has fewer L2 misses and thus the prediction error is negative.

5 Predicting Energy for Different *f* Points

Our methodology in predicting the dynamic energy of a program is the following: we measure static power (power in idle state) under all available different frequencies. To

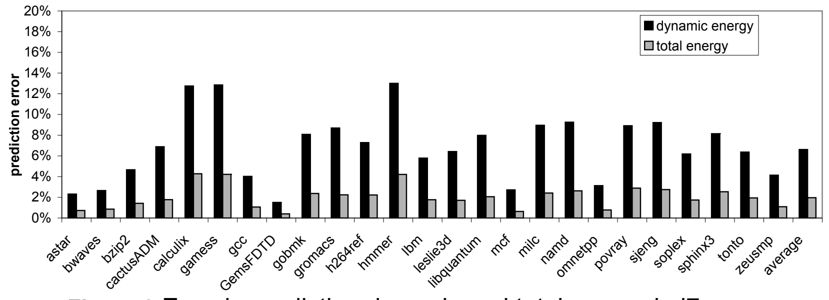


Figure 3 Error in predicting dynamic and total energy in i7 core.

calculate the run-time dynamic energy of a program, we subtract from the total energy the product of execution time and static power for the corresponding frequency. Finally, to predict the dynamic energy consumed in a new frequency, we tested the two extremes of a fully clock-gated and fully non-clock-gated processor: dynamic energy in the former case is proportional to the square of the voltage ($E \sim V^2$), while in the latter case the energy should be computed according to the formula: $E \sim f * V^2 * t$.

Our experimental findings reveal that the i7 core is not highly clock-gated, since the fully non-clock-gating scenario produced better results. Especially, in memory bound programs, in which the clock gating is expected to save more energy due to the long stall intervals, the results for the fully clock-gated case were even worse compared to the non clock gated scenario. A more accurate model could be derived if we knew exactly the processor clock gating map. However this information is not available [3]. Total energy can be predicted by adding to the predicted dynamic energy the product of the new static power and the execution time. Figure 3 shows our results. The grey bar shows the results for total (dynamic and static) energy prediction, while for clarity reasons we also depict the results for dynamic energy prediction (black bar). Figure 3 depicts the errors when all cores are active. Maximum and average error are 14% and 8% respectively for dynamic energy prediction, while the errors in total energy prediction are less than 5% and 2% respectively. In case that fewer cores were activated from BIOS, the dynamic energy prediction error would be the same but the total energy prediction error would be between the current value and the dynamic energy error for each program.

6 Overview of the Methodology and Future Work

The study presented in the paper shows a development stage of our work in building a strong framework for power/performance runtime management for the Intel processors. Our view is that this framework can form the basis for future power-related research. A unique characteristic of our approach (compared to previous approaches) is that it requires minimal input and calculations. The required inputs are: a single performance counter (line 1 in Table 1) and the power consumed by the processor collected by monitoring the off-chip voltage regulator. Both inputs are gathered at run-time by our kernel module. The idle power consumed by the processor in all frequencies (9 values in i7) is also stored in our kernel module. Based on those inputs, our kernel module is able to predict the effect of frequency scaling with minimal calculations (presented in our previous work [9]). The whole approach runs in kernel space thus introducing minimal timing overhead (less than 1%) in the execution of the applications.

In this work, we focus on the following scenario: we perform a single whole run of an application and using our methodology we predict the execution time and energy consumed by the application in different frequencies (i.e. the core frequency is kept constant during the whole execution). We are currently extending this work towards a window-based approach in which our kernel module applies different V/f points at runtime aiming to optimize different energy-efficient metrics (e.g EDP, application/OS energy metrics) in analogy to our runtime DVFS management in a simulated environment [9].

7 Conclusions

We described a hardware-specific implementation of the stall-based model proposed in our previous work [9]. In order to explain how the model performs in the i7 core, we attempted a qualitative analysis of how prediction accuracy is affected by various benchmarks' behavior. Our experimental results show that the execution time of the applications can be predicted for various frequency scaling steps —even for the extreme scaling from *fmax* to *fmin*— by our model with good accuracy. We also reverse engineer the power behavior of i7 core by measuring static energy dissipation, as well as dynamic energy consumed during the execution of programs and predicting how the power consumed by the processor is affected by frequency scaling.

Acknowledgments—This work is supported by the EU-FP7 ICT Projects, "A highly efficient adaptive multi-processor framework (HEAP)," Contract No. 247615, and "Embedded Reconfigurable Architecture (ERA)," Contract No. 249059.

8 References

[1] S. Eyerman, et al. A mechanistic performance model for superscalar out-of-order processors. Transactions on Computer Systems, 2010.

[2] S. Eyerman, et al. A performance counter architecture for computing accurate CPI components. Int. Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

[3] Intel Core™ i7-800 and i5-700 Desktop Processor Series, Intel, 2010.

[4] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. Annual Int. Symposium on Microarchitecture, 2006.

[5] C. Isci, et al. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. Int. Symposium on Microarchitecture, 2006.

[6] C. Isci and M.Martonosi. Run-time power monitoring in high-end processors: methodology and empirical data. Annual Int. Symposium on Microarchitecture, 2003.

[7] T. Karkhanis and J.E. Smith. A first-order superscalar processor model. Annual Int. Symposium on Computer Architecture, 2004.

[8] S. Kaxiras and M. Martonosi. Computer architecture techniques for power-efficiency. Morgan & Claypool Publishers, 2008.

[9] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time DVFS orchestration in superscalar processors. Int. Conference on Computer Frontiers, 2010.

[10] G. Maglis, et al. Profile-based dynamic power voltage and frequency scaling for a multiple clock domain processor. Int. Conference on Computer Architecture, 2003.

[11] M. C. Maury, et al. Online power-performance adaptation of multithreaded programs using event-based prediction. Int. Conference on Supercomputing, 2006.

[12] M. C. Maury, et al. Prediction-based power-performance adaptation of multithreaded scientific codes. Transactions on Parallel and Distributed Systems, 2008.

A Sleep-based Communication Mechanism to Save Processor Utilization in Distributed Streaming Systems

Shoaib Akram and Angelos Bilas[†]

Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {shbakram,bilas}@ics.forth.gr

Abstract. Energy-efficiency of applications deployed in data-centers is becoming increasingly important. Techniques that reduce CPU utilization for specific workloads can help improve energy consumption. An application domain that has been studied in the past extensively and is lately gaining importance in data-centers is distributed stream processing. In this work we examine an existing stream processing system, Borealis [6], and we identify significant sources of overhead in the communication stack. Specifically, we examine the inter-node communication path in a distributed setup and the overheads associated as streams flow from node to node. We find that the send and receive tasks in Borealis take up significant CPU resources. We redesign the send and receive paths of Borealis by replacing TCP with a user-level protocol based on Myrinet MX. We then evaluate the CPU utilization and network throughput on a 10 Gbits/s network using both polling and interrupts for communicating data. Finally, we propose a sleep mechanism that avoids the CPU overheads associated with both interrupts and polling. We use a real setup consisting of four eight-core nodes equipped with 10 Gbits/s Ethernet and native Myrinet MX communication subsystems to examine the impact of our approach. Our results show that our approach saves CPU utilization for a range of workload conditions and is able to achieve better throughput compared to TCP with lower CPU utilization (upto 40%).

1 Introduction

Modern data-center applications tend to employ complex software stacks that are processor hungry. Recent work [7, 18] shows that CPU utilization can be directly correlated to total system energy consumption.

Stream processing is a workload that has been studied extensively in the past and has recently been gaining attention due the data-oriented nature of many

[†]Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR-71409, Greece.

modern applications. In this work we examine the impact of the communication protocol on CPU utilization of distributed stream processing applications over 10 Gbits/s networks. We use an existing stream processing system, Borealis [6], to investigate how improved communication mechanisms can reduce CPU utilization.

Traditionally, in communication-based applications, senders and receivers need to detect the arrival of new (data or control) messages. Two approaches to detecting these events are using interrupts or polling [17]. Neither of these approaches is satisfactory. On one hand, interrupts incur high overhead at high-network speeds and large packet rates. Polling, on the other hand, wastes CPU at low packet rates. In addition, previous work has also examined adaptive techniques that switch between interrupts and polling [13].

An alternative approach is to release the CPU by sleeping for a specific amount of time when required events are still pending. Compared to interrupts, releasing the CPU can be implemented synchronously thus avoiding overheads related to asynchronous events. However, this approach can result in waking up long before or after the event is due. Compared to polling, sleeping can release the CPU to other threads but results in a system call. Ultimately, sleeping has the potential to adjust CPU utilization to the required processing rate. The main obstacle is regulating the sleep interval.

One approach to address this problem is to try and predict the amount of sleep that is appropriate at any given point. However, in modern operating systems it is not possible to exactly regulate sleep time at fine grain, because it depends on a number of coarse grain events in the operating system kernel. Thus, a more robust approach is to use a notion of doing work in "waves". Consider two nodes connected in a pipeline fashion with the first node being the sender and the second node being the receiver. The receiver informs the sender of the available memory buffers that it has reserved for receiving data and then checks the buffers for the arrival of data. If the test fails, the receiver sleeps for a fixed amount of time. In essence, the receiver accumulates work while taking no extra CPU cycles. Similarly, the sender distributes the work and then waits for a message from the receiver. This message informs the sender if the receiver is ready to receive more data. If the test for the message fails, the sender sleeps for a fixed amount of time. Thus, the sender accumulates work for the receiver, which will process this during the next "wave". The relationship between network throughput and processing rate determines the exact shape of these "waves".

Normally in applications such as borealis, there is a separate task that process the incoming data. Thus one way to look at our approach is that the sleep gives the opportunity to conserve energy by not consuming excess energy polling or processing interrupts. This more of the available CPU time is spent doing useful work.

Implementing such a "wave" approach requires dealing with three issues. First, we need to convert blocking send and receive operations to non-blocking combined with a sleep operation. Second, we need to introduce an appropriate buffering mechanism at each of these points to allow the rest of the system to operate during the sleep. Finally, we need to ensure that the amount of buffering available at any point is appropriate for tolerating the inaccuracy of the corresponding sleep operation.

We investigate the impact of this approach on Borealis. The original version of Borealis uses TCP/IP for inter-node communication. It is possible to evaluate our sleep-based approach with a communication stack based on TCP. However, we port the communication stack of borealis to user-level Myrinet MX for two reasons. First, as we will show in Section 3, user-level communication protocols result in higher throughput compared to TCP/IP. This helps to accomodate the drop in throughput due to the difficulty of implementing a precise sleep interval for an entire range of system parameters. Further, a user-level communication protocol allows a fine-grained control of buffering resources in the send and receive paths since all the memory buffers for sending and receiving are allocated at the user-layer.

In this work, we create a version of Borealis that replaces kernel-based TCP/IP with user-level Myrinet MX communication [5]. Our modified version of Borealis uses a more light-weight communication mechanism and is able to achieve higher throughput than the original, TCP/IP-based version. We implement all three approaches (interrupts, polling, sleeping) in this version of Borealis and examine the impact on CPU utilization. We also compare the results with the original Borealis with TCP. We use a setup of four, eight-core systems (with each one being 2-way hyperthreaded) connected with a 10 Gbits/s Myrinet network using simple streaming workloads.

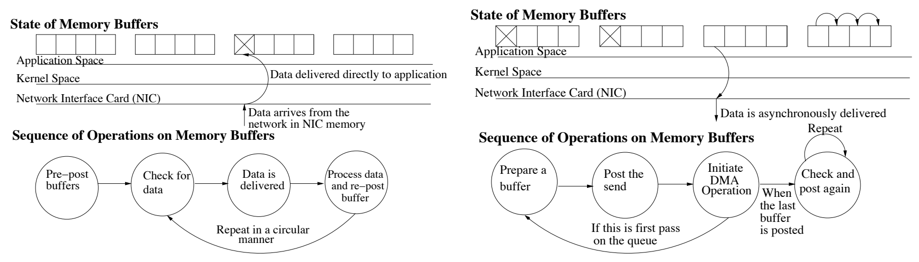
The rest of this paper is organized as follows. Section 2 presents our design of the new communication subsystem of Borealis using Myrinet MX and the new sleeping mechanism that saves CPU utilization for a range of workload conditions. Section 3 presents our experimental platform and evaluation methodology and discusses our experimental results. In Section 4 we present related work. Finally, we draw our conclusions in Section 5.

2 Communication Subsystem Design

In this section, we describe our modifications to the communication subsystem of Borealis and our proposed algorithm for sleeping in the send and receive threads.

2.1 Communication using Myrinet MX

Figure 1 shows our implementation of Borealis using Myrinet MX. During the initialization phase, the receiver allocates and posts a number of buffers where data could be received directly from the network interface controller (NIC). Later, the receiver can poll each location to find out if data has arrived in the buffer. If so, data is picked up from the buffer and the buffer is posted again as ready for reception of new data. Similarly, the sender operates on a queue of buffers. First, the sender fills up and post all buffers of the queue. Then, the


State of Memory Buffers
Application Space
Kernel Space
Network Interface Card (NIC)
Data delivered directly to application
Data arrives from the network in NIC memory
Sequence of Operations on Memory Buffers
Pre–post buffers
Check for data
Data is delivered
Process data and re–post buffer
Repeat in a circular manner
State of Memory Buffers
Application Space
Kernel Space
Network Interface Card (NIC)
Data is asynchronously delivered
Sequence of Operations on Memory Buffers
Prepare a buffer
Post the send
Initiate DMA Operation
Check and post again
Repeat
When the last buffer is posted
If this is first pass on the queue

(a) MX receiver (b) MX sender

Fig. 1. MX-based communication in Borealis. 'X' indicates that the buffer is holding valid data and is not available for re-use. Arrows indicate the sequence in which operations are performed.

sender scans the queue one by one, checks for the completion of DMA and fills up and posts the buffer for sending data.

In Myrinet MX there is a need to deal with "unexpected" messages [15]. Myrinet MX, similar to other user-level communication systems, operates without copies when receive buffers for messages have already been pre-posted. In case a message arrives at the receiver and there is no receive buffer specified by the application the system will deliver the message to a library-level, internal buffer. Later, when the message is successfully detected via a receive operation, it is copied to the application buffer. To ensure that Myrinet MX will operate without data copies in the receive path, there is a need to ensure that receive buffers are pre-posted and arriving messages are always delivered directly to these application buffers. For this purpose we use an application-level flow control mechanism for buffer management purposes between the sender and the receiver. The receiver (Figure 1(a)) pre-posts an agreed number of buffers for sender to send events. Then, the receiver updates the sender with new credits as it frees receive buffers after sending events to another thread that processes these events.

2.2 Sleep-based communication algorithms

We use non-blocking send and receive Myrinet MX calls combined with sleep system calls to replace blocking send and receive calls. This in turn requires introducing buffering at certain points in the path to ensure that other parts of the system continue processing when a specific thread sleeps. Figure 1 and Algorithms 1 show our receive and send paths.

In the receive path, we check each buffer for arrival of data. If the test fails, the receive thread sleeps for a fixed amount of time. In the send path we check for completion of a DMA operation on a buffer posted earlier. If the operation is not complete yet, the send thread sleeps for a fixed amount of time. Also, before posting a send buffer, the sender checks to see if credits are available for sending data. If there are no credits to send data, the send thread sleeps for a

Algorithm 1 Receive (left) and send (right) path with sleeping enabled.

```
n=size of circular queue that holds incoming or outgoing events
f=frequency with which to send credits to upstream node
t=time interval for which to sleep
m=number of credits to send to the upstream node
Receiver:
// initialize receive buffers and credits
post_buffers($n$)
send_credits($n$);
$i = 0$
// wait until next receive buffer contains new event
while $TRUE$ do
  while poll_buffer($i$) do
    sleep($t$)
  end while
  process_event($i$)
  $i = (i + 1)\%n$
  // if above threshold replenish sender
  if $(i\%f) = (f - 1)$ then
    send_credits(m)
  end if
end while

Sender:
credits=waitfor_credits()
$i = 0$
while $TRUE$ do
  //wait for next send buffer
  while poll_buffer($i$) do
    sleep($t$)
  end while
  prepare_event($i$)
  //replenish in case you run out
  credits += collect_credits()
  while $credits = 0$ do
    sleep($t$)
    credits += collect_credits()
  end while
  send_event($i$)
  $i = (i + 1)\%n$
end while
```

fixed amount of time. The sleep in the receive thread and the sleep when there are no credits available are related and work together to reduce CPU utilization without reducing overall throughput.

The rational behind sleeping in the receive thread is to accumulate work while consuming no CPU cycles. While the receive thread sleeps it has already posted buffers for receiving data. When the receiver wakes up, it takes some time to process buffers and re-post them. Therefore, the credits to receive more data are sent to the upstream node after some time which depends upon the frequency with which credits are sent. The send thread of the node upstream sleeps during this time interval to conserve energy. If the sleeping intervals are approximately correct, when the send thread wakes up, it will receive credits to send data to the downstream node.

The minimum sleep time in our systems is 2 ms. However, we use a sleep interval of 10 ms in our sleeping algorithms. We experimentally find this to be an appropriate time interval for our specific systems. Note that the sleep interval can change based upon the system load. An algorithm to adjust sleep interval according to system load is left as future work.

3 Experimental results

In this section, we describe our evaluation methodology, the experimental platform, and our experimental results.

3.1 Experimental platform and methodology

Our test environment consists of four, server-type systems running the Linux operating system (CentOS release 5.4). Each system has two Intel Xeon Quad-core chipsets. The cores have two-way hyper-threaded capability. Therefore, each machine can potentially execute sixteen threads simultaneously. Each machine has 14 Gbytes of DRAM physical memory and a 10 Gbits/s Ethernet NIC from Myricom that is capable of operating both in TCP/IP and Myrinet MX mode. The four machines are physically connected via 10 Gbits/s Ethernet HP ProCurve 3400cl switch. The first node in the pipeline runs a load-generator that generates a batch of tuples (events). The next two nodes in the pipeline run Borealis. The last node runs a light-weight receiver that receives tuples and consumes them internally without storing them.

We use an in-house micro-benchmark that consists of two filter operators in a chain. The parameters of the filter are set to pass each incoming tuple down the pipeline.

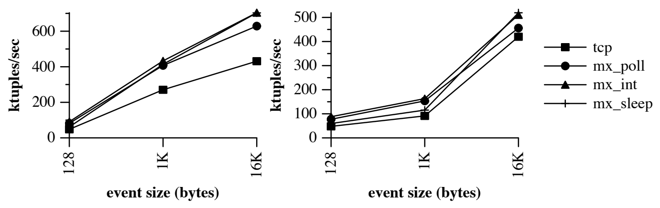
Our main goal is to understand the impact on performance and CPU utilization of using spinning, interrupts, and sleeping in the send and receive threads of Borealis with Myrinet MX API vs. TCP/IP. We build and use the four configurations of Borealis, *tcp, mx-poll, mx-int, mx-sleep*, that use TCP, Myrinet MX with interrupts, polling, and our sleeping mechanism, respectively.

We perform experiments with 8 instances of Borealis running on each node. A different stream is associated with each instance of Borealis. There is a separate load-generator (source) and receiver (sink) for each instance. A given instance of Borealis can not utilize the entire bandwidth of 10 Gbits/s Ethernet because processing time becomes the limiting factor. For this reason, we use multiple instances of Borealis on each node to exploit the maximum bandwidth available from the underlying 10 Gbits/s network. We believe that this is a realistic mode of operation for data streaming systems. Finally, we show results for different tuple sizes and different batching factors.

We perform experiments for different tuple sizes and event sizes. Note that a larger event size is obtained by using a large batching factor. We use a buffer queue of 100 entries on the receive side and send credits to the nodes upstream after processing every 10 buffers. The size of queue on the send side is 10 entries. The sender posts a send operation for receiving credits when 5 credits are left.

3.2 Impact on Network Throughput

Figure 2 shows the network throughput of Borealis for TCP and Myrinet MX (interrupts, polling, and sleeping). In all our experiments, the processing thread is the bottleneck. For this reason the use of polling when running 8 instances of Borealis results in a lower throughput compared to that of interrupts. Using Myrinet MX with interrupts instead of TCP improves the throughput of Borealis by upto 22% for a tuple size of 512 bytes. Further note that mx-adp always results in better throughput compared to TCP between $23 - 63\%$.



(a) 128 bytes (b) 512 bytes

Fig. 2. Network throughput of Borealis in tuples/s for TCP/IP and Myrinet MX for different tuple and batch (event) sizes running a filter query.

3.3 Impact on CPU Utilization

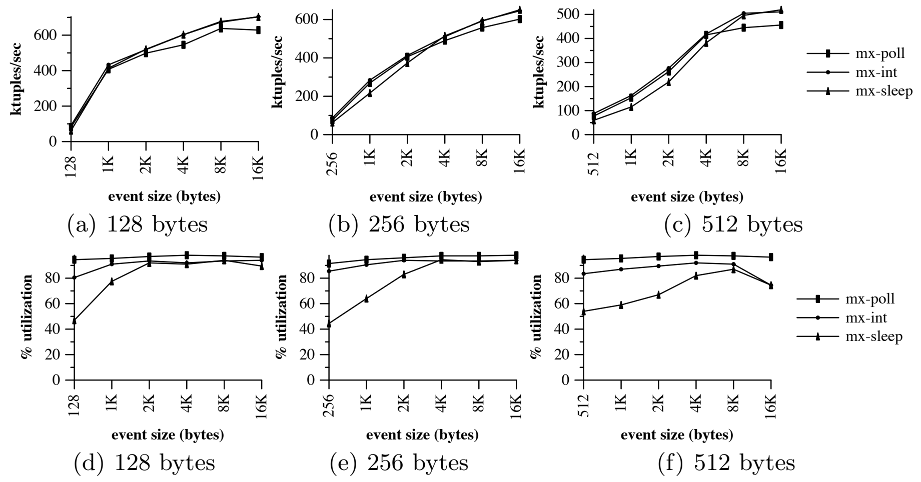
Figure 3(d-e) shows the CPU utilization of Borealis for Myrinet MX (polling, interrupts, and sleeping). CPU utilization is reported as the average utilization across the two nodes that run Borealis. A utilization of 100% in Figure 3(d-e) means that all 16 hardware threads in both nodes are fully utilized. We note that in all cases CPU utilization exceeds 90% for both TCP (not shown) and Myrinet MX with polling. Using Myrinet MX with interrupts reduces the CPU utilization slightly. Note that at small event sizes, the receive thread does not have enough work to perform as it receives fewer tuples per event, quickly enqueues them, and waits for the next batch. Therefore, mx-adp results in significant reduction in CPU utilization compared to both mx-poll and mx-int. For an event size of 1024 bytes (batching factor of 8), mx-adp approaches exactly the throughput of mx-int, with saving in CPU utilization upto 15% for a tuple size of 128. Note that since data streaming is an upcoming application, the typical range of parameters such as batching factor is not obvious. However, since our sleep-based mechanisms do not result in significant loss in throughput compared to mx-int, we believe it to be a reasonable approach.

Next, we note that we do not observe any saving in CPU utilization beyond event sizes of 1024 bytes, 2048 bytes and 4096 bytes respectively for tuple size of 128 bytes, 256 bytes and 512 bytes. This is because events larger than these result in a large batching factor. A large batching factor implies more tuples in a single event and thus high overhead relating to enqueuing of tuples compared to communication overhead of sending or receiving a single event. Therefore, when the receive thread enqueues one event, it finds the next event readily available.

In summary, an implementation based upon Myrinet MX with sleep-based send and receive paths, we are able to achieve better throughput compared to TCP and reduce CPU utilization upto 40%.

4 Related Work

Data streaming has recently been gaining importance due to the large number of existing and emerging applications that need to process data [10]. Research both in academia [6, 9, 3] and industry [16] aims at building scalable distributed



(a) 128 bytes (b) 256 bytes (c) 512 bytes

(d) 128 bytes (e) 256 bytes (f) 512 bytes

Fig. 3. Network throughput in tuples/s and % utilization for different tuple and batch (event) sizes for a filter query.

stream processing systems. Efforts span the space from designing and implementing efficient relational operators for streaming databases [1], to proposing high-level query languages for specifying streaming workloads [8, 4], and to mapping different applications to streaming systems [20, 11, 1].

Less attention has so far been paid to understanding the performance implications of communication protocols and infrastructure for this communication-intensive class of applications. The authors in [16] presents an evaluation of SystemS, a data streaming system built by IBM. They discuss at a high level the impact of communication on streaming performance. In contrast, in our work we not only quantify the performance of an existing stream processing system on a cluster consisting of modern server machines but also discuss a number of specific issues related to the communication protocol stack and related optimizations. We also evaluate in detail the impact of these aspects and show how future streaming systems can benefit from careful design.

Recently there has also been increased interest in research on issues related to (in)efficiency of software stacks in data center applications. The authors in [14] examine trends in building data center applications from existing components that lead to large inefficiencies. In addition, recent work has been pointing out that software stack inefficiencies have an impact on the energy efficiency of data center infrastructures [2, 12]. These approaches propose architectural techniques to adaptively manage power subject to changing workload conditions or policies above the hardware layer to exploit techniques and have pointed out inefficiencies in the software stack of existing middleware systems.

The authors in [19] have proposed hardware mechanisms to avoid OS spin overheads. Their techniques addresses the problem of extra spin overhead in over-committed virtual machines running operating systems that do not use gang scheduling. In this paper, we quantify the negative impact of using spinning in the send and receive paths of the communication stack of Borealis when the system and underlying network is heavily-loaded.

5 Conclusions

In this work we examine how the communication stack of a distributed streaming system, Borealis, can use a sleep-based mechanism to avoid both interrupt and polling overheads. We identify subtle reasons for excessive CPU utilization in this class of applications and propose mechanisms to improve the efficiency of individual nodes. We propose a sleeping mechanism that saves CPU utilization for a range of workload conditions. In comparison with TCP, using Myrinet MX and our sleep-based send and receive operations achieves better throughput and reduces CPU utilization upto 40% for a range of parameters. We expect that the importance of our approach will increase as systems become more heterogeneous by combining networks and processors of different speeds in consolidated data center environments that execute multiple distributed applications at the same time..

6 Acknowledgments

We would like to thank Manolis Marazakis for his help with designing and implementing the base Myrinet MX version of Borealis. We thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs through the STREAM (FP7-ICT-216181), HiPEAC (NoE-004408), HiPEAC2 (FP7-ICT-217068), and SCALUS (FP7-PEOPLE-ITN-2008-238808) projects. We thank Jim Holt for useful feedback on an initial version of this paper.

References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] Eric Anderson and Joseph Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44(1):40–45, 2010.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur etintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[7] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 13–23, New York, NY, USA, 2007. ACM.

[8] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[9] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Patrick Valduriez. Streamcloud: A large scale data streaming system. In *ICDCS*, pages 126–137. IEEE Computer Society, 2010.

[10] Julian Hyde. Data in flight. *Queue*, 7(11):20–26, 2009.

[11] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: optimizing stream processing applications via graph partitioning. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[12] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, 2010.

[13] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling watchdog: combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 179–188, New York, NY, USA, 1996. ACM.

[14] N. Mitchell. The big pileup. pages 1 –1, mar. 2010.

[15] Myrinet Inc. Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, Version 1.2. http://www.myri.com/scs/MX/doc/mx.pdf, October 01, 2006.

[16] Toyotaro Suzumura, Toshiaki Yasue, and Tamiya Onodera. Scalable performance of system s for extract-transform-load processing. In *SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–14, New York, NY, USA, 2010. ACM.

[17] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[18] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. Worth their watts? - an empirical study of datacenter servers. pages 1 –10, jan. 2010.

[19] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 124–133, New York, NY, USA, 2006. ACM.

[20] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*, pages 306–325, Berlin, Heidelberg, 2008. Springer-Verlag.