Overlapping Computation and Communication in SMT Clusters with Commodity Interconnects

Georgios Goumas, Nikos Anastopoulos, Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Athens, Greece
HiPEAC Members
Email: {goumas,anastop,nkoziris}@cslab.ece.ntua.gr

Nikolas Ioannou
University of Edinburgh
School of Informatics
Edinburgh, United Kingdom
HiPEAC Member
Email: nikolas.ioannou@ed.ac.uk

***Abstract*—In this paper we focus on optimizing the performance in a cluster of Simultaneous Multithreading (SMT) processors connected with a commodity interconnect (e.g. Gbit Ethernet), by applying overlapping of computation with communication. As a test case we consider the parallelized advection equation and discuss the steps that need to be followed to semantically allow overlapping to occur. We propose an implementation based on the concept of Helper Threading that distributes computation and communication in the two sibling threads of an SMT processor, thus creating an asymmetric pair of execution patterns in each hardware context. Our experimental results in an 8-node cluster interconnected with commodity Gbit Ethernet demonstrate that the proposed implementation is able to achieve substantial performance improvements that can exceed 20% in some cases, by efficiently utilizing the available resources of the SMT processors.**

***Keywords*-SMT architecture; overlapping;**

I. INTRODUCTION

Simultaneous multithreading (SMT) [1] provides a promising strategy to increase the throughput of conventional superscalar processors by multiplexing the execution of concurrent threads. It performs a sparing duplication or expansion of certain processor functional units (e.g. processor architecture state), thus targeting noteworthy performance improvement at minimal additional construction cost. Depending on the architectural design, the processor's functional units may be duplicated, dynamically shared or statically partitioned. The main goal is to fill up the frequent empty issue slots that are due to low Instruction Level Parallelism (ILP) or long-latency events such as cache misses and branch mispredictions, with operations executed by an alternative thread. Unfortunately, SMTs suffer from contention for the common resources. In fact SMTs have been shown to be beneficial only for threads that utilize different functional resources, e.g. server applications.

Attaining performance for a single application running on an SMT processor is quite an intricate task. Traditional parallelization approaches that lead to the construction of identical threads operating on partitioned data do not seem promising, since in this case threads will compete for the same functional units. Yet, if the sequential code is unoptimized in terms of memory access, then one can expect an efficient multiplexing of executing threads that will fill the numerous cache-miss stalls. In general, however, *Thread-Level Parallelism (TLP)* is not as suitable for SMTs as for general Symmetric Multiprocessing (SMP) architectures as experimentally verified in [2]. For this reason researchers have tried to utilize hardware threads of a single application in an asymmetric fashion (e.g. [3], [4]), frequently named *Helper Threading*. In the practical case of an SMT with two hardware threads, Helper Threading dictates that the second thread should perform some useful but different work from the main computation thread. The most interesting example of Helper Threading is Speculative Precomputation [5], [6], in which the helper thread precomputes memory accesses on behalf of the main computation thread, attacking in this way possible bottlenecks due to memory latency [7], [8].

Overlapping computation and communication is an important optimization for message-passing applications. This capability is a key characteristic of modern, high-speed interconnects like Inifiniband [9], Myrinet [10] and Quadrics [11], [12]. Software implementations that incorporate this feature always assume such an underlying interconnect [13]–[15], which is able to offload communication operations from the main CPU to the intelligent logic of the NIC. The main ambition of this paper is to relax this assumption by enabling the operation of this execution model even over a commodity interconnection network. Instead of overlapping computation and communication in the interconnection, we opt for a much simpler and cost efficient solution, where we do so by correctly splitting our program and using existing SMTs. More specifically, we re-formulate the problem so as to split it in two threads, the computation one and the communication one, both of which run concurrently on the SMT. The key intuition here is that since the two threads utilize different resources, they generally do not interfere with each other, resulting in an efficient execution of both.

Unfortunately, the primary step in accomplishing overlap between computation and communication involves careful

re-engineering of the application. The main aspect is how to design the two threads such that they execute concurrently while respecting the inter-thread dependences [13]–[15]. In subsequent sections we show the rationale behind the required re-engineering and apply it to a stencil computation arising from the discretization of the advection partial differential equation (PDE). We discuss linear scheduling for this family of computations and select a proper linear scheduling vector that theoretically allows overlap of computation and communication. Since our interconnection network does not support overlapping, we need to assign these two tasks to the two available threads of an SMT processor, and properly synchronize them. This task distribution creates two substantially asymmetric threads, the one performing floating point computations and the other performing memory copies and communication operations. Thus we expect this scheme to provide an efficient utilization of the SMT processor resources. Indeed, our experimental results demonstrate that the proposed implementation makes a very efficient use of our platform resources and can provide a non-negligible performance improvement exceeding 20%.

The rest of the paper is organized as follows: the next section provides background knowledge, while Section III discusses the overlapping linear schedule used in the proposed implementation of advection PDE proposed in Section IV. In Section V we present experimental results that compare the performance of various implementations, while in Section VI we discuss previous, related work. Finally, this paper concludes in Section VII.

II. BACKGROUND

A. Basic concepts of SMT architecture

SMT allows a conventional dynamic, superscalar processor to issue instructions from multiple independent threads in a single cycle. The key motivation in this technique comes from the observation that in single-thread execution of many applications, a considerable portion of the processor's issue bandwidth remains unutilized. This is due to the insufficient ILP inherent in many applications that leaves multiple issue slots unused in each cycle, or due to long latency operations, such as cache misses and branch mispredictions, that stall the entire pipeline and leave all issue slots unused for successive cycles. In either case, instructions from alternative threads can be scheduled, filling up empty issue slots. In essence, by fetching instructions from additional threads the SMT logic provides the out-of-order engine with a window containing many more non-dependent instructions, which increases the scheduling opportunities of the engine and maximizes resource utilization. Note that after the renaming stage, the out-of-order engine of the SMT processor is oblivious to logical processor distinctions. Therefore, with SMT, thread level parallelism is effectively converted to instruction level parallelism.

The additional hardware needed to provide a conventional processor with SMT capabilities is minimal. For example, in the first implementations of Intel's Hyper-threaded processors it accounted for less than 5% of the total chip area. Only those structures necessary to track independently each logical processor's execution are replicated, most important being the program counters and register mapping tables (i.e. the architectural state). All other resources are either statically partitioned (e.g. intermediate micro-op queues, load/store buffers, the reorder buffer in Hyper-threaded processors), or dynamically shared (e.g. execution units, caches, branch predictor, control logic and buses). For this reason, it is argued that mutual exclusion in the use of these resources is an important requirement for achieving high multithreaded performance. Threads with heterogeneous instruction mixes and complementary resource needs (e.g., memory-intensive vs. computation-intensive, fp-bound vs. integer-bound, etc.) may coexist well under simultaneous execution. On the other hand, threads with symmetric profiles tend to compete for the same execution units in each cycle. This creates conflicts, pipeline stalls, and finally performance degradation.

B. Algorithmic model and advection equation

Our algorithmic model concerns applications that involve $(n+1)$-dimensional perfectly nested loops with constant dependences. The iteration space J^{n+1} is rectangular, thus it holds $J^{n+1} = \{\vec{j}(j_1, j_2, \ldots, j_{n+1}) \in Z^{n+1} \wedge; l_i \leq j_i \leq u_i, ; i = 1 \ldots n+1\}$, where $l_i, u_i \in Z$ are the lower and upper bounds of the i-th loop respectively. The dependences of the problem are expressed with constant, $(n+1)$-dimensional dependence vectors $\vec{d_i}$, $i = 1 \ldots m$. We denote $\vec{d_{ij}}$ the j-th element of vector $\vec{d_i}$. In the class of problems under consideration it holds $\vec{d_{ij}} \geq 0$, $i = 1 \ldots m$ and $j = 1 \ldots n+1$. The dependence matrix of the algorithm, denoted D, is an $(n+1) \times m$ matrix containing as columns the dependence vectors of the algorithm. Overall, the algorithms have the general form of Algorithm 1, where U is an $(n+1)$-dimensional array and F is a linear function.

Algorithm 1 algorithmic model

```
1: for j_1 ← l_1 to u_1 do
2:   ...
3:   for j_n ← l_n to u_n do
4:     for j_{n+1} ← l_{n+1} to u_{n+1} do
5:       U[j] = F(U[j − d_1], ..., U[j − d_m]);
6:     end for
7:   end for
8:   ...
9: end for
```

Discretization of the advection equation leads to an application following our algorithmic model. Advection is the physical process of transportation within a fluid described by the PDE $\frac{\partial v}{\partial t} = \vec{a} \bigtriangledown v$ where v is particle density or

temperature and $\vec{a}$ is the vector field, e.g. the velocity vector of the material. In one spatial dimension the above equation is equivalent to:

$$\frac{\partial v}{\partial t} = a\frac{\partial v}{\partial x} \tag{1}$$

If we need to study an advection process in a space with length X for a time window T, we can discretize the initial domain into a uniform grid using a time step Δt and a space step Δx. Then, we can discretize the above PDE using a variety of finite differencing schemes. For example, if we employ the *Euler-Forward* scheme [16], the time derivative can be substituted by a fraction of differences as follows: $\frac{\partial v}{\partial t} = \frac{v_i^{n+1} - v_i^n}{\Delta t}$. The physics of the problem allows us to employ *upwind* [16] differencing schemes for the space derivative, which involves computations with "previous" spatial grid points. Thus, in this case we can substitute the space partial derivative as follows: $\frac{\partial v}{\partial x} = \frac{v_i^n - v_{i-1}^n}{\Delta x}$. If we substitute to Equation (1) we get $v_i^{n+1} = \left(1 + a\frac{\Delta t}{\Delta x}\right) v_i^n - a\frac{\Delta t}{\Delta x} v_{i-1}^n$. If we exploit the serial traversal of the above equation, we can utilize previous spatial elements computed at the current time step as shown in the following equation:

$$v_i^{n+1} = \left(1 + a\frac{\Delta t}{\Delta x}\right) v_i^n - a\frac{\Delta t}{\Delta x} v_{i-1}^{n+1} \tag{2}$$

Note that v_i^0 and v_0^n are known from the initial and boundary values of the PDE problem. Equation (2) can be easily solved for all points in the discretized computational grid $T' \times X'$ where $T' = T/\ \Delta t$ and $X' = X/\Delta x$ with the nested loop shown in Algorithm 2.

Algorithm 2 nested loop for 1-D advection equation

for $j_1 \leftarrow 0$ to T' **do**
  for $j_2 \leftarrow 1$ to X' **do**
    $U[j_1+1][j_2] = (1+a \cdot dt/dx) \cdot U[j_1][j_2] - a \cdot dt/dx \cdot U[j_1+1][j_2-1]$;
  end for
end for

The dependence matrix of the above algorithm is $D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. The discretization process followed leads to nonnegative elements in the dependence matrix.

C. Tiling, scheduling and mapping

The algorithmic dependences of the applications under consideration enable us to apply rectangular tiling [17], [18] in order to tune the granularity of communication. As far as the scheduling of tiles is concerned, we apply linear scheduling techniques [19]. Central to linear scheduling is the notion of the scheduling vector Π. Intuitively, in our class of applications, it suffices to calculate the inner product of a point $\vec{j} \in J^{n+1}$ with Π to derive the parallel time step at which $\vec{j}$ will be executed. Π is legal iff $\Pi\vec{d_i} > 0$, $i = 1 \ldots m$. The most efficient scheduling vector is the one that minimizes the total parallel execution time, i.e. the one that minimizes the execution time of the lexicographically largest point of the iteration space.

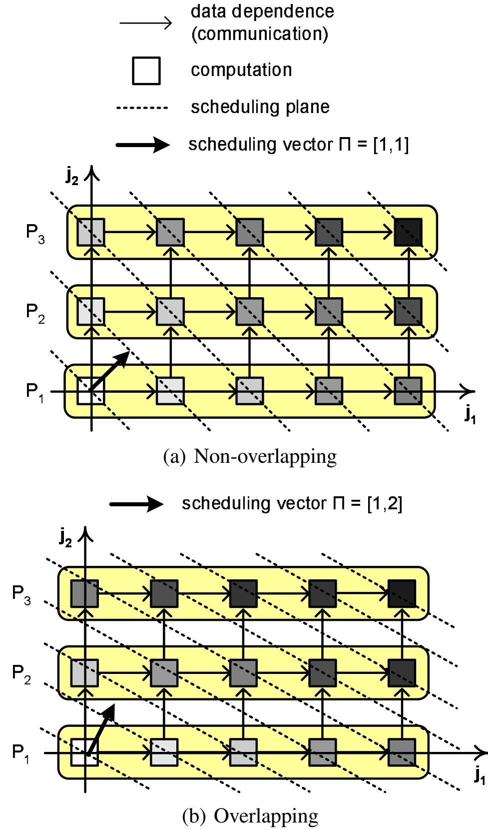


Figure 1: Linear schedules for a 2-dimensional iteration space (1-D advection).

All points (or tiles) that lie within each n-dimensional surface perpendicular to the scheduling vector Π can execute in parallel, thus, one can employ an n-dimensional array of processes to maximize parallelism [20]. In our approach we will also consider the general case of an n-dimensional process grid to execute in parallel $(n+1)$-dimensional iteration (or tiled) spaces.

III. Overlapping vs. Non-overlapping Linear Scheduling

The first step to achieve computation and communication overlap is to re-engineer the application. If the most efficient in terms of minimal parallel steps is employed, then computation/communication overlapping is not possible. This is so because the data dependences force each processor to perform sequentially the following steps: wait for the communication data, perform computations on the data received and send computed data to its neighbors. This is depicted for the 2-dimensional iteration space in Figure 1a, which corresponds to the 1-D advection equation. The rectangles representing computation can be either single iteration points

Figure 2: Non-overlapping implementation for parallel advection with two symmetric threads.

or tiles. In this case we employ a 1-dimensional processor array and all computations along the first dimension are mapped to the same processor. The straightforward linear schedule for the dependences of our problem that minimizes the total parallel execution steps [19] is $\Pi = [1, 1]$ ($\Pi = [1, 1, \ldots, 1]$ in general). However, note from Figure 1a that in this case each processor needs to wait for communication data from its predecessor, then perform computations and finally forward the newly computed data to its successor. The execution proceeds in distinct parallel phases of computations and communications. This clearly, does not allow for any communication to computation overlap.

However, as shown in [21], one can encapsulate both communication and computation in each step at the cost of theoretically increasing the total number of parallel execution steps. This can be achieved in terms of linear scheduling by the use of a vector $\Pi = [2, 2, \ldots 1, \ldots 2, 2]$, where we apply 1 only along the mapping dimension (the dimension along which all computation is assigned to the same processor). This overlapping schedule is depicted in Figure 1b for a 2-dimensional space. In this case each processor is able to concurrently do the following: perform computations for the current time step, send data computed at the previous time step and receive data that will be used during the next time step. The total number of parallel time steps is indeed increased but each step in this case involves both computation and communication (all computations and communications cut by the same parallel plane can be executed in parallel). This theoretical overlapping is realistic only when the underlying platform enables communication to be offloaded from the main CPU.

IV. Implementation details

In this section we provide implementation details for the proposed overlapping scheme. Our execution platform is a cluster of SMT nodes, with each SMT having two hardware threads. The underlying interconnection does not provide overlapping capabilities, e.g. it is a commodity Gbit Ethernet. Algorithm 3 presents the pseudocode of the standard, non-overlapping 1-D advection using MPI primitives. The general form of the code remains conceptually the same for the more interesting 2-D (3-D) cases, with the only difference that in these cases a central MPI process needs to reveive from 2 (3) and send to 2 (3) neighbors. To utilize all available hardware threads of the platform, one can assign two symmetric threads in the SMT of each node, as shown in Figure 2, where it is clear that communication and computation is performed in distinct non-overlapped phases.

Algorithm 3 Code snippet for non-overlapping 1-D advection

```
1: while steps < K do
2:     MPI_Irecv(down,...); {init receive from downward neighbor}
3:     MPI_Wait(...); {wait receive completion}
4:     UnpackData(steps); {unpack data for current step}
5:     Compute(steps); {computations of current step}
6:     PackData(steps); {pack data of current step}
7:     MPI_Isend(up,...); {init send to upward neighbor}
8:     MPI_Wait(...); {wait send completion}
9:     steps++;
10: end while
```

Algorithm 4 Code snippet for overlapping 1-D advection with symmetric threads

```
1: while steps < K do
2:     MPI_Irecv(down,...); {init receive from downward neighbor}
3:     MPI_Isend(up,...); {init send to upward neighbor}
4:     Compute(steps); {computations of current step}
5:     MPI_Waitall(...); {wait communication completion}
6:     UnpackData(steps+1); {unpack data for next step}
7:     PackData(steps); {pack data for next step's send}
       steps++;
8: end while
```

The second implementation we consider is that of the straightforward overlapping scheme. The pseudocode is shown in Algorithm 4. In this case the communication is initiated before the computations and is performed in an overlapped fashion with them, under the assumption that the interconnection network is able to offload communication operations (e.g. memory copies, OS traps, communication protocol, polling etc.) from the main CPU (e.g. [22]). This execution pattern is also demonstrated in Figure 3. The steps of computation and communication in this case are multiplexed, since receptions are performed for data to be used in the next step, and sends are performed for data calculated in the previous step.

In our case the communication network does not support overlapping, but we can distribute computation and communication between the two threads of an SMT processor,

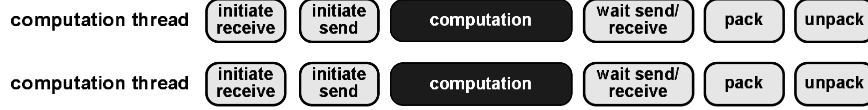


Figure 3: Overlapping implementation for parallel advection with symmetric threads.

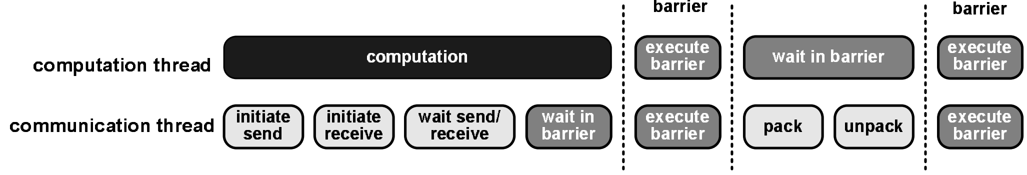


Figure 4: Overlapping implementation for parallel advection with asymmetric threads.

Algorithm 5 Code snippet for overlapping 1-D advection with asymmetric threads

1: {computation thread}
2: **while** steps $<$ K **do**
3: Compute(steps); {computations of current step}
4: barrier(); {synchronize after end of computation}
5: barrier(); {synchronize after end of communication}
6: steps++;
7: **end while**

8: {communication thread}
9: **while** steps $<$ K **do**
10: MPI_Irecv(down,...); {initiate receive from downward neighbor}
11: MPI_Isend(up,...); {initiate send to upward neighbor}
12: MPI_Waitall(...); {wait communications completion}
13: barrier(); {synchronize after end of computation}
14: UnpackData(steps+1); {unpack data for next step}
15: PackData(steps); {pack data for next step's send}
16: barrier(); {synchronize after end of communication}
17: steps++;
18: **end while**

as shown in Algorithm 5 and Figure 4. The computation thread undertakes only the floating-point operations while all the communication tasks (packing/unpacking, MPI function calls) are offloaded to the communication thread. Additional synchronization primitives (e.g. barriers) are also required to correctly orchestrate the execution of the two threads and preserve the semantics of the algorithm. In particular, the computation thread needs to signal the end of the computations in each step, while the communication thread signals the end of data packing and unpacking, to proceed to the next execution step.

V. Experiments

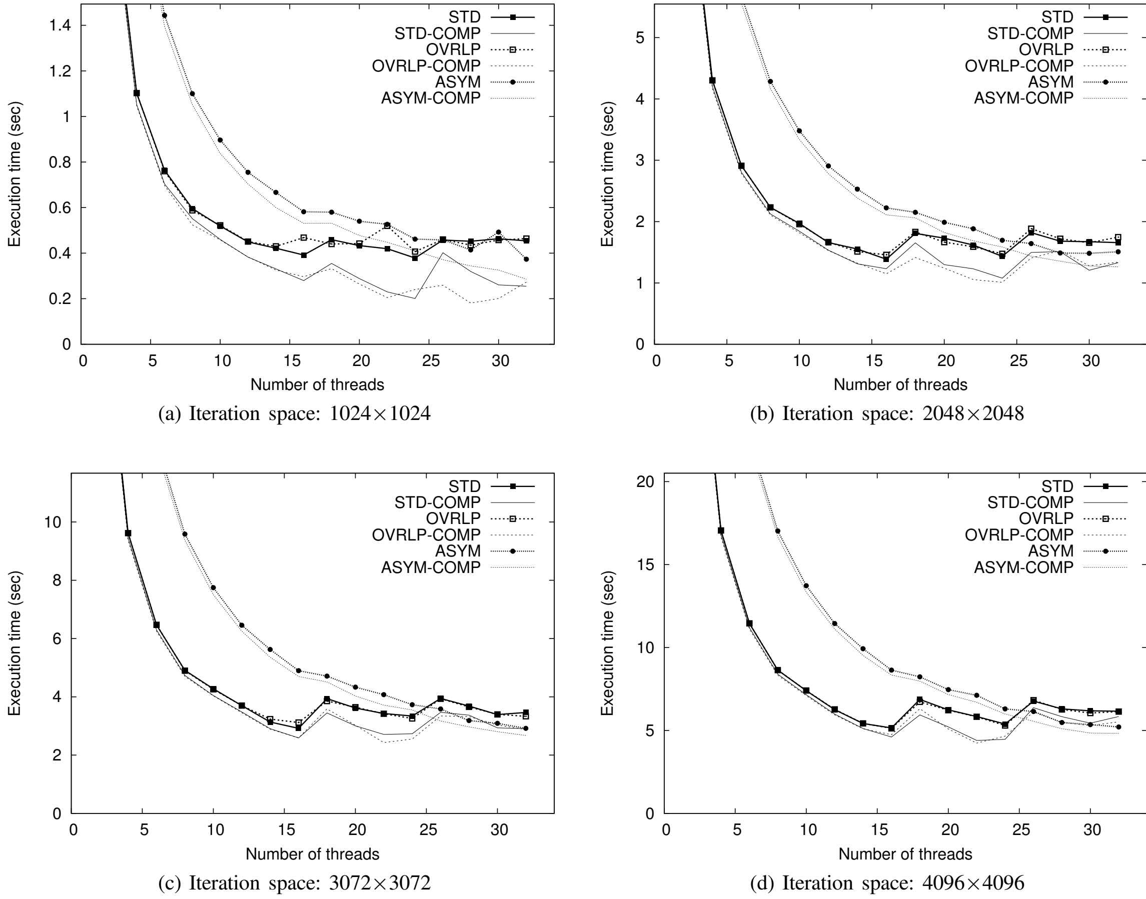
In this section we evaluate the efficiency of the three parallel implementations of the advection equation discusssed in Section IV in a cluster of SMT processors. The versions we consider are: the standard, non-overlapping (STD) shown in Algorithm 3 and Figure 2, the overlapping with symmetrical threads (OVRLP) shown in Algorithm 4 and Figure 3, and the proposed overlapping with asymmetrical threads (ASYM) shown in Algorithm 5 and Figure 4.

A. Experimental setup

Our execution platform is an 8-node cluster of SMT processors (xenon1–xenon8). Each node contains two Intel Xeon processors running at 2.8GHz, with 2GB of main memory and 1MB L2 cache. The processors are enabled with HT technology, that makes a single physical processor appear as two logical processors by applying a two-threaded SMT approach. The OS (Linux with 2.6 kernel) identifies two different logical processors, each maintaining a separate run queue. Thus, in total the platform has up to 32 logical processors. The nodes are interconnected with commodity Gbit Ethernet adapters over a single 24-port switch.

We have implemented the two-dimensional (2-D) and three-dimensional (3-D) advection equation and parallelized them using MPICH version 1.2.7. We tune the granularity of communication with tiling and tested several candidate tile sizes. In the 2-D (3-D) case, we applied a two-dimensional (three-dimensional) process topology. Since, as shown in [23], the number of processes in each dimension of the process topology also affects the communication overheads, we experimented with all possible process topologies. In each case we report the best performance attained. Double precision arithmetic is applied.

In the case of STD and OVRLP we assigned all even numbers of MPI processes between 2 and 32 in a cyclic fashion, i.e. we first fill all cluster nodes with one thread, then assign a second thread to the second SMT processor of the node, and finally start to involve HT in the execution. For example, 20 threads (MPI processes) are running in our cluster in the following way: xenon1–xenon4 are assigned 3 MPI processes each, with one processor fully utilizing its hardware threads, while xenon4–xenon8 are assigned 2 MPI processes allocated to the two separate packages of the nodes. From that point, adding couples of MPI processes starts involving HT in two additional nodes. In the case of



(a) Iteration space: 1024×1024

(b) Iteration space: 2048×2048

(c) Iteration space: 3072×3072

(d) Iteration space: 4096×4096

Figure 5: Overall parallel execution time for the three versions of 2-D advection: standard, non-overlapping (STD), overlapping with symmetric threads (OVRLP) and overlapping with asymmetric threads (ASYM). The relevant execution time in each case is also demonstrated (*-COMP).

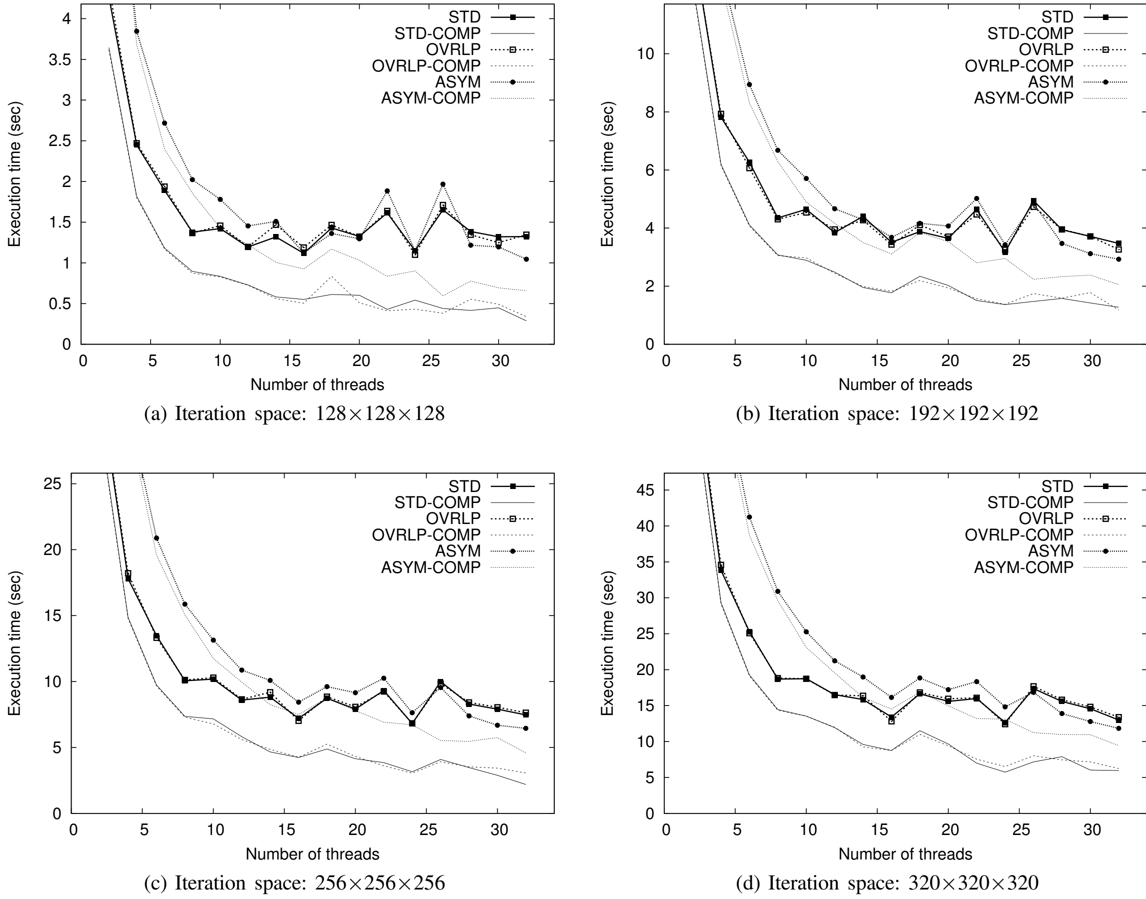
ASYM we use POSIX threads to spawn one computation and one communication thread for each MPI process, assign these threads to the same package and use the primitives provided by the Pthreads library to properly synchronize them. Thus, in this case we need 16 MPI processes to fill the execution platform with 32 threads.

B. Results for 2-D advection

Figure 5 shows the experimental results (overall parallel execution time and computation time) of 2-D advection for the parallel implementations under consideration in four iteration spaces. In all experiments the third dimension of the iteration space (number of parallel execution steps for the untiled version) was set to 256. Several interesting observations can be made. At first we can see that the STD and OVRLP implementations have similar performance behavior. This verifies that although OVRLP conceptually implements overlapping, the underlying hardware is not capable of hiding any part of the communication overhead. On the other hand, the overlapping schedule presented in Section III does not cause any performance overheads, thus it leads to a viable implementation, regardless of the underlying communication hardware.

Both implementations (STD and OVRLP) exhibit good scalability until 16 threads. Beyond that number of threads, nodes start to involve HT which clearly seems to have a negative impact on performance. This is expected since both implementations have symmetric threads competing for the same functional units of the SMT processor (see Section II-A). The proposed ASYM implementation scales well until 32 threads. As expected, for smaller number of threads the performance of ASYM is worse than that of STD and OVRLP, since in this case ASYM does not use all the potential of the underlying platform. For example, in 16 threads STD and OVRLP utilize all 16 processors of the cluster, while ASYM uses only 8. Beyond 16 threads, the three implementations start to converge with ASYM finally outperforming the other two when all 32 logical processors of the cluster are involved.

Overall, all three implementations have a similar record in



(a) Iteration space: 128×128×128

(b) Iteration space: 192×192×192

(c) Iteration space: 256×256×256

(d) Iteration space: 320×320×320

Figure 6: Overall parallel execution time for the three versions of 3-D advection: standard, non-overlapping (STD), overlapping with symmetric threads (OVRLP) and overlapping with asymmetric threads (ASYM). The relevant execution time in each case is also demonstrated (*-COMP).

their achievements concerning the most efficient utilization of the underlying execution platform. Comparing the performance for 16 threads in STD and OVRLP with 32 threads in ASYM, the average differences are smaller than 3%. The proposed ASYM implementation is not able to provide any meaningful performance improvements, since as seen from the difference between the computation time (*-COMP) and overall execution times, the communication overhead in 2-D advection is very low. Since the goal of ASYM is to hide communication under computation, this is not expected to lead to performance improvements in this class of problems. Note, however that in the small iteration space (1024×1024) where the data fit in the L2 cache, making communication a larger fraction in the overall parallel execution time, ASYM shows a capability to hide communication leading to a slight performance improvement of 5%.

C. Results for 3-D advection

In 3-D advection the communication overhead is much larger since in this case each process needs to perform communication with six neighboring ones. Figure 6 shows the experimental results of 3-D advection for the parallel implementations under consideration in four iteration spaces. Again here, the fourth dimension of the iteration space (number of parallel execution steps for the untiled version) was set to 256. Concerning the performance behavior and scalability, the observations are similar to the 2-D advection: STD and overlap perform in similar ways and scale well until 16 threads. Beyond that number of threads, due to the involvement of HT, their performance drops. Note, however, that in this case the difference between the overall parallel execution and the computation time is very high, indicating that the communication overhead represents a crucial fraction of the overall time. In this case we notice that the ASYM implementation succeeds very well in hiding the communication, a fact that leads to non-negligible overall performance improvements. Comparing the best scores of each implementation (16 threads in STD and OVRLP and 32 threads in ASYM) ASYM is able to provide on average a 13% overall performance improvement over STD and can

reach up to 20% (in iteration space 192×192×192). Further performance comparisons between the three implementations are provided in the next paragraph.

D. Overall evaluation

In order to gain a better insight into the capabilities and shortcomings of each implementation, we provide the plots of Figure 7, where we depict the normalized (to the overall parallel execution time of STD) computation, communication and parallel execution times of OVRLP and ASYM. Figures 7a and 7b demonstrate the capabilities of the three implementations to utilize all the available resources of our 8-node cluster, i.e. by assigning threads in all 32 logical processors. As expected, the ASYM implementation based on the assignment of asymmetric threads on the same SMT processor leads to an average 17% and 18% performance improvement in 2-D and 3-D respectively.

However, the most important metric to assess the three implementations, is their maximum performance for the given platform. This is achieved using 16 threads in STD and OVRLP and 32 threads in ASYM. The results are visualized in Figure 7c for 3-D only, since in 2-D we noted no significant differences between the three implementations. 3-D ASYM is able to provide on average a 13% overall performance improvement over STD and can reach up to 20% (in iteration space 192×192×192) by successfully reducing the overhead of communication.

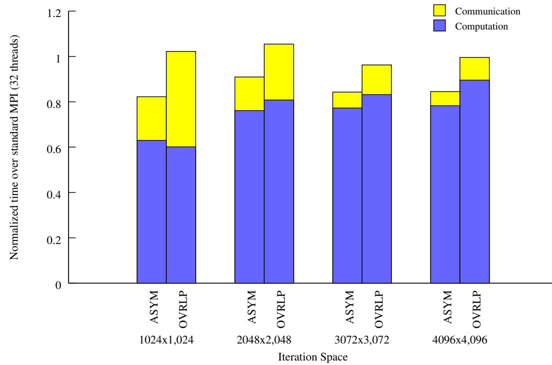
A third comparison seems also quite interesting: In this case we consider 8 threads for STD and OVRLP, and 16 threads for ASYM, that provides a solid view of the performance for an 8-node, single-processor cluster. Since the interference of HT is harmful for STD and OVRLP, using 8 threads is the best configuration for these implementations. The comparison in this case is shown in Figure 7d. On average, ASYM outperforms STD in this execution environment by 19% and can reach up to 23% performance improvement (in iteration space 128×128×128).

Overall, we can conclude that in the presence of significant communication overheads, as is the case with 3-D advection, the proposed ASYM implementation is capable of achieving non-negligible performance improvements compared to the standard and straightforward overlapping implementation. Indeed, ASYM offloads communication operations to the sibling thread of SMT and exploiting the asymmetry of the computation and communication threads reaches up to a 20% performance improvement in a two-processor cluster and up to 23% in a single-processor cluster. This is a noteworthy performance improvement since, as reported in [2], HT can provide a performance boost of 20-30% in the processor used. Our experimental results also demonstrate that ASYM greatly reduces communication overheads, although it increases the overall computation time (Figures 5–7). This can be attributed to the fact that ASYM uses half the processors for computations used by STD and OVRLP. Finally, as depicted in Figure 4, ASYM has to also pay the additional cost of synchronization between the communication and computation threads. As discussed in [24], this overhead is significant in SMT processors.

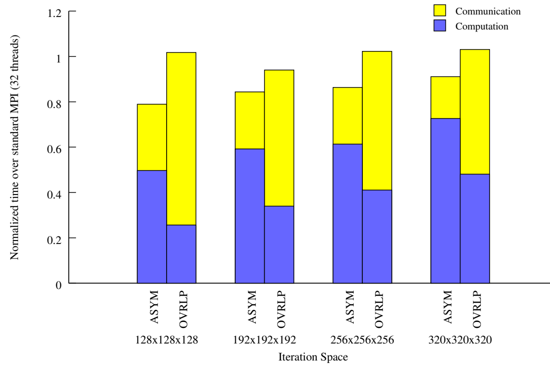
VI. Related work

The first step to achieve overlapping of computation with communication is to break the inherent serialization of an applications, by re-engineering the algorithm itself. For example, Lain and Banerjee [25] work on irregular stencil computations and propose a graph coloring scheme to enable communications to occur in parallel with computations. Cerio et al. [26] propose a technique named communication pipelining, according to which data are communicated as soon as they are produced. The authors consider hypercube networks with asynchronous communication protocols supporting overlapping. In [21] the overlapping schedule for the class of applications under consideration was proposed, while in [13] this schedule was coupled with an underlying SCI network [27] to implement actual overlapping. Danalis et al. [14] present a method to transform MPI programs directed towards improving communication-computation overlap in MPI collective operations. The approach is verified using Myrinet. Bell et al. [15] focus on FFT as a test application, and show how overlapping can be beneficial with the use of small messages that perform communication as soon as data are ready. Their experimental platform involve clusters with Infiniband, Quadrics and Myrinet. Quite recently Sancho and Kerbyson [28] multiplex several Conjugate Gradient solvers to achieve overlapping in a cluster interconnected with Infiniband.

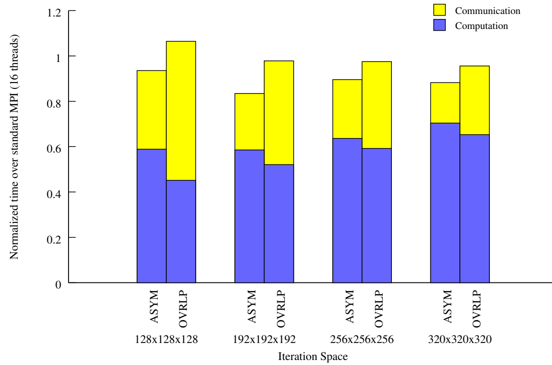
Asymmetric threading scenarios for Hyper-threaded processors have been explored also in several contexts. Prefetching helper threads [7], [8]. run along the main application thread on an idle hardware context and speculatively prefetch data into a shared cache, following a technique known as Speculative Precomputation. These schemes rely on earlier software-controlled helper threading schemes, studied in the context of simulated SMT models [4], [6]. The key difference is that the latter assumed idealized hardware support (e.g., ideal SMT implementation, multiple spare contexts for the helper threads, special hardware for thread management, etc.) which turned out to be a determinant factor for achieving good performance. Researchers have also proposed hardware-controlled helper threading schemes, where a number of helper threads invisible to software transparently perform cache prefetching or optimize branch predictions on behalf of the main application thread [3], [29], [30]. In [31] helper threading is incorporated within Dijkstra's algorithm and coupled with Transactional Memory to speed up parallel execution. Zhang et. al [32] propose a dynamic optimization framework in which the helper threads dynamically optimize the code of the application thread as it executes. Gummaraju and Rosenblum in [33] investigate the



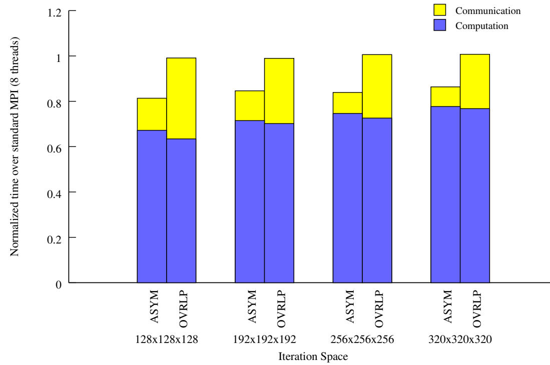
(a) 2D: 32 threads in STD/OVRLP/ASYM



(b) 3D: 32 threads in STD/OVRLP/ASYM



(c) 3D: 32 threads in ASYM, 16 threads in STD/OVRLP



(d) 3D: 16 threads in ASYM, 8 threads in STD/OVRLP

Figure 7: Comparison between the three implementations for various combinations of threads.

mapping of stream programs on a Hyper-threaded processor. This work is relevant to ours in the sense that the stream programming paradigm provides a way to decouple memory accesses and computation in a program, thus being a good case for SMT processors. The authors propose separate work queues for each kind of operation to dynamically overlap memory operations and computations.

VII. Conclusions

In this paper we worked on the efficient implementation of computation to communication overlap in a cluster of SMTs with a commodity interconnect. Since our interconnection network does not support overlapping, our goal is to offload communication operations to the sibling thread of an SMT processor. In this way, we are able to assign two asymmetric threads to the processor, a strategy that is proven to be beneficial for this architectural design. Our test case is the parallelized advection equation, which is executed based on a proper linear schedule that allows concurrent computation and communication phases. Our experimental results demonstrate that the proposed implementation is able to provide non-negligible performance improvements that can exceed 20% compared to the standard non-overlapping parallelization scheme.

Acknowledgments

This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (80%) and National Resources (20%).

References

[1] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.

[2] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading pentium 4 processor," in *PACT '03: Proc. of the 12th international conference on Parallel architectures and compilation techniques*. Washington, DC, USA: IEEE Computer Society, 2003, p. 26.

[3] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," *ISCA '99: Proc. of the 26th annual international symposium on Computer architecture*, no. 2, pp. 186–195, 1999.

[4] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *ISCA '01: Proc. of the 28th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2001, pp. 40–51.

[5] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *ISCA '01: Proc. of the 28th annual international symposium on Computer*

architecture. New York, NY, USA: ACM Press, 2001, pp. 14–25.

[6] H. Wang, P. H. Wang, S. M. Ettinger, S. S. wei Liao, and J. P. Shen, “Speculative precomputation: Exploring the use of multithreading for latency,” in *Intel Technology Journal*, vol. 6, no. 1, 2002. [Online]. Available: citeseer.ist.psu.edu/586671.html

[7] D. Kim, S. S. wei Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, “Physical experimentation with prefetching helper threads on intel’s hyper-threaded processors,” in *CGO ’04: Proc. of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2004, p. 27.

[8] E. Athanasaki, N. Anastopoulos, K. Kourtis, and N. Koziris, “Exploring the performance limits of simultaneous multithreading for memory intensive applications,” *J. Supercomput.*, vol. 44, no. 1, pp. 64–97, 2008.

[9] I. T. Association, “InfiniBand Architecture Specification, Release 1.0, 2000,” http://www.infinibandta.org/specs.

[10] 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*, vol. 15, no. 1, pp. 29–36, Feb 1995.

[11] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, “Quadrics network (qsnet): High-performance clustering technology,” in *Hot Interconnects 9*, Stanford University, Palo Alto, CA, August 2001.

[12] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, “Performance comparison of mpi implementations over infiniband, myrinet and quadrics,” in *SC ’03: Proc. of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 58.

[13] N. Koziris, A. Sotiropoulos, and G. Goumas, “A Pipelined Schedule to Minimize Completion Time for Loop Tiling with Computation and Communication Overlapping,” *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1138–1151, Nov 2003.

[14] A. Danalis, K. Y. Kim, L. Pollock, and M. Swany, “Transformations to parallel codes for communication-computation overlap,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC05)*, Seattle, WA, USA, November 2005.

[15] C. Bell, D. Bonachea, R. Nishtala, and K.Yelick, “Optimizing Bandwidth Limited Problems using One-sided Communication and Overlap,” in *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS’06)*, Rhodes, Greece, Apr 2006.

[16] G. E. Karniadakis and R. M. Kirby, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, 2002.

[17] F. Irigoin and R. Triolet, “Supernode Partitioning,” in *Proc. of the 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages (POPL’85)*, San Diego, California, USA, Jan 1988, pp. 319–329.

[18] J. Ramanujam and P. Sadayappan, “Tiling Multidimensional Iteration Spaces for Multicomputers,” *Journal of Parallel and Distributed Computing*, vol. 16, pp. 108–120, 1992.

[19] W. Shang and J. Fortes, “Time Optimal Linear Schedules for Algorithms with Uniform Dependencies,” *IEEE Trans. on Computers*, vol. 40, no. 6, pp. 723–742, 1991.

[20] W. Shang and J. Fortes, “On Time Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 3, pp. 350–363, 1992.

[21] G. Goumas, A. Sotiropoulos, and N. Koziris, “Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping,” in *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS’01)*, San Francisco, USA, Apr 2001.

[22] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, “Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits,” in *PPoPP ’06: Proc. of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 32–39.

[23] G. Goumas, N. Drosinos, and N. Koziris, “Communication-aware supernode shape,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 498–511, 2009.

[24] N. Anastopoulos and N. Koziris, “Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors,” in *Proc. of the 2nd Workshop on Multithreaded Architectures and Applications (MTAAP 2008)*, Mar 2008.

[25] A. Lain and P. Banerjee, “Techniques to overlap computation and communication in irregular iterative applications,” in *ICS ’94: Proc. of the 8th international conference on Supercomputing*. New York, NY, USA: ACM, 1994, pp. 236–245.

[26] L. Díz de Cerio, M. Valero-García, and A. González, “A method for exploiting communication/computation overlap in hypercubes,” *Parallel Comput.*, vol. 24, no. 2, pp. 221–245, 1998.

[27] H. Hellwagner, “The SCI Standard and Applications of SCI,” in *Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters*, H. Hellwagner and A. Reinefield, Eds. Springer-Verlag, Sep 1999, pp. 3–34.

[28] J. C. Sancho and D. J. Kerbyson, “Improving the performance of multiple conjugate gradient solvers by exploiting overlap,” in *Euro-Par ’08: Proc. of the 14th international Euro-Par conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 688–697.

[29] A. Roth and G. S. Sohi, “Speculative data-driven multithreading,” in *HPCA ’01: Proc. of the IEEE 7th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2001, p. 37.

[30] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” in *ISCA ’01: Proc. of the 28th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2001, pp. 2–13.

[31] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris, “Employing Transactional Memory and Helper Threads to Speedup Dijkstra’s Algorithm,” in *Proc. of the 38th International Conference on Parallel Processing (ICPP 2009)*, Sep. 2009.

[32] W. Zhang, B. Calder, and D. Tullsen, “An event-driven multithreaded dynamic optimization framework,” in *PACT 05: Proc. of the 14th international conference on Parallel Architectures and Compilation Techniques*, 2005, pp. 350–360.

[33] J. Gummaraju and M. Rosenblum, “Stream programming on general-purpose processors,” in *MICRO 38: Proc. of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 343–354.