Coalition Threading: Combining Traditional and Non-Traditional Parallelism to Maximize Scalability

Md Kamruzzaman
Computer Science and Engineering
University of California San Diego
mkamruzz@cs.ucsd.edu

Steven Swanson
Computer Science and Engineering
University of California San Diego
swanson@cs.ucsd.edu

Dean M. Tullsen
Computer Science and Engineering
University of California San Diego
tullsen@cs.ucsd.edu

ABSTRACT

Non-traditional parallelism provides parallel speedup for a single thread without the need to manually divide and coordinate computation. This paper describes *coalition threading*, a technique that seeks the ideal combination of traditional and non-traditional threading to make the best use of available hardware parallelism. Coalition threading provides up to 2× gains over traditional parallel techniques on individual loops. However, deciding when and to what degree to apply either traditional or non-traditional threading is a difficult decision. This paper provides heuristics for identifying loops that benefit from a combination of traditional and non-traditional parallelism and those that will perform best with a single technique. Using this heuristic, coalition threading provides an average gain of 17% across all the loops and an average speedup of 16.7% for the full applications over traditional parallelism. This performance is within 0.7% of the speedup that an oracle heuristic could attain.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors–Optimization

General Terms

Languages, Performance

Keywords

non-traditional parallelism, inter-core prefetching, chip multiprocessors, helper threads, compilers

1. INTRODUCTION

In the past decade, multiprocessor architectures have gone from being the domain of high-end servers and specialized high-performance computing systems, to becoming ubiquitous in every type of computing platform from smart phones to the data center. While these architectures have delivered the potential for scalable, parallel performance, the impact at the application level has been uneven – software parallelism is not nearly as pervasive as hardware parallelism. There are some environments that provide near-infinite parallelism. Other environments and workloads scale poorly or not at all. If we are to continue to provide performance scaling in this era of increasing hardware parallelism, we must scale application performance both in the presence of abundant software parallelism and when that parallelism is harder to find.

Traditional parallel execution techniques involve the programmer, compiler, libraries, or language constructs dividing the computation and memory activity of an application across processing elements to achieve speedup. However, many applications do not naturally allow this type of division of effort and even fewer return perfect performance scaling when they are so divided.

Researchers have proposed several *non-traditional parallelism (NTP)* techniques to provide parallel speedup (several cores or hardware contexts execute a program faster than a single core) without changing the number of logical threads (e.g., pthread or MPI threads) running the computation. Non-traditional parallelism techniques include helper thread prefetching [5, 7, 12, 13, 14, 16], helper thread dynamic recompilation [6, 15, 27, 28], and software data spreading [11]. Speculative multithreading (thread-level speculation) [18, 21, 23] maintains single-thread semantics from a software perspective, but tends to look more like traditional parallelism when running on the hardware.

Previous work demonstrates that non-traditional parallelism is useful when traditional parallelism is not available (i.e., serial code). This paper shows that it is also effective even when traditional parallelism exists. For many loops, the best parallel solution is to apply non-traditional parallelism on top of traditional parallelism rather than using only traditional parallelism. We can exploit this by using a combination of these two types of parallelism that applies different techniques to different parts of an application to increase scalability. We call this combination *coalition threading*, and in this paper we examine, in particular, the combination of traditional parallelism expressed as pthreads and OpenMP constructs, paired with inter-core prefetching (ICP) [12]. ICP uses a helper thread to prefetch data into a local cache on one core, then migrates execution to that core. This technique requires no hardware support and works across cores, sockets, and SMT contexts, so it is applicable to a wide range of current and future parallel microprocessors.

This paper quantifies the benefits of coalition threading on a state-of-the-art multiprocessor, running a range of benchmarks, and describes compiler heuristics that can identify the loops in each benchmark most suited to non-traditional techniques. While coalition threading represents an opportunity for the programmer or compiler to increase parallel speedup and scalability, deciding when to apply each technique is not easy, and making a poor decision can adversely impact performance.

Our results show that coalition threading with ICP can outperform traditional threading in 39% of the 108 loops in eleven applications we studied, chosen to only include those that already saw some benefit from traditional parallelism. By selecting the best threading strategy on a loop-by-loop basis, we demonstrate that coalition threading improves total performance by 17% on average and up to 51% for some applications.

Achieving that level of performance requires that the compiler, programmer, or runtime system be able to determine when to apply each type of parallelism. We find that a standard machine learning technique, linear classification [8], accurately classifies 98% of the loops where the decision is critical and 87% overall. Considering application performance, the heuristic achieves over 99.4% of the performance that an oracle could deliver.

This paper makes the following contributions. It is the first investigation into the effectiveness of combining non-traditional parallelization techniques with traditional parallelization. It demonstrates that a combined approach can as much as double the performance of traditional parallelization alone for particular loops. We show that performance can be highly sensitive to the decision made for each loop. To handle that, this paper develops heuristics that a compiler can easily implement to identify which combination of traditional and non-traditional parallelism will work best.

The remainder of this paper is organized as follows. Section 2 discusses work related to this research. Section 3 describes coalition threading. Section 4 describes our methodology. Section 5 compares the performance of parallelization techniques. Section 6 derives and evaluates heuristics that allow the compiler to accurately identify the best parallel approach, and Section 7 concludes.

2. BACKGROUND AND RELATED WORK

This work combines elements of traditional parallelism and helper thread prefetching. This section describes prior work in these two areas.

Traditional Parallelism.

This research specifically focuses on explicitly programmed parallelism, and a set of applications known to achieve parallel speedup by using Pthreads [19] or OpenMP [4]. Coalition threading, however, could be applied to most forms of traditional parallelism, including mechanisms such as MPI that do not communicate through shared memory.

Non-traditional Parallelism.

Non-traditional parallelism encompasses several existing techniques including helper threads, dynamic compilation, and some types of speculative threading.

Helper threads execute in parallel with the main thread, accelerating the application without necessarily taking over any of the computation [5]. Helper threads can be used to precompute memory addresses for prefetching [5, 6, 7, 10, 12, 13, 14, 15, 16, 25], to precompute branch outcomes [28], and to recompile hot traces in the main thread based on runtime behavior [26, 27]. This work uses helper threads for data prefetching.

Most of the prior work on helper-thread prefetching targets multithreaded (e.g., simultaneous multithreaded, or SMT) processors and executes prefetch code in another SMT context [5, 6, 7, 13, 14, 16, 28]. Other techniques execute helper threads in separate cores in a CMP [10, 12, 15]. All these techniques "distill" the main thread code to include just enough computation to predict future load addresses.

Systems construct helper threads in different ways including hardware-only schemes [6], conventional profile-driven compilers [13], hand-construction [7, 12, 28], and dynamic compilation [15, 27]. This work does not address helper thread creation, but assumes they exist.

This research particularly exploits Inter-Core Prefetching [12], a software technique that uses thread migrations and allows helper thread prefetching on multicores that do not share the lowest level caches. Unlike SMT-based helper threads, ICP can exploit any type of hardware parallelism between cores, sockets, or thread contexts. When applied across cores, it reduces competition for execution resources and private cache space between the main thread and the helper threads.

Other techniques have applied NTP to CMPs. Data Spreading [11] partitions a thread's working set across the system's caches rather than prefetching. Slipstream processors [10] execute a reduced version of the program ahead of the main program in another core to provide fault tolerance and improve performance.

Speculative multithreading [18, 21, 23] straddles the line between traditional and non-traditional parallelism. Most of this work strives to maintain single-thread software semantics, while the hardware executes threads in parallel. A typical example of speculative multithreading would speculatively execute loop iterations from the serial execution across multiple cores or SMT contexts and verify correctness at runtime. Mitosis [21] is a hardware-software approach that extends the concept further by adding helper threads to precompute critical data.

Other Related Work.

Researchers have also improved scalability of parallel code by combining different types of traditional parallelism and by dynamically changing thread behavior. Researchers have proposed hybrid models [17] that combine subsets of Pthreads, OpenMP, and MPI as well as systems that combine multiple forms of parallelism. For example, Gordon, et al. [9] propose a compiler that combines task parallelism, data parallelism, and pipeline parallelism for stream programming. Feedback-driven threading [24] shows, analytically, how to dynamically control the number of threads to improve performance and power consumption. It dynamically measures data synchronization and bus bandwidth usage to modify its threading strategy.

3. COALITION THREADING

Coalition threading augments conventional parallel threads with helper threads to improve performance and scalability. Coalition threading works because many loops (virtually all, if you scale far enough) have limited scalability – non-traditional parallelism can pick up when traditional tails off. In addition, NTP is not bound by the limits of traditional parallelism. Recent work [11, 12] has demonstrated greater than n-fold speedups with n helper threads.

This means that NTP can be a profitable choice even for scalable loops.

In principle, coalition threading will work with many different kinds of helper threads, but we focus on inter-core prefetching [12], a software-only technique that works on any cache coherent system and subsumes other techniques like software data spreading [11], SMT helper thread prefetching [7], and shared cache prefetching [15].

The following section describes ICP and identifies the key issues that arise in coalition threading in general and with ICP in particular. Then we describe the implementation of our coalition threading framework.

3.1 Inter-core prefetching

ICP helper threads prefetch data into their local caches, paving the way for main threads that come behind and perform the program's actual computation. To accomplish this, ICP divides the execution of a loop into multiple *chunks*. The main thread executes the program on one core while a helper thread runs concurrently on another, prefetching a future chunk into the local cache. Once the main thread finishes its chunk, the threads swap cores. The main thread finds the data it needs waiting for it and the helper thread starts prefetching another chunk.

Like other helper-thread prefetchers, inter-core prefetching is most useful when the access pattern is somewhat non-linear (e.g., accessing an array of pointers or multi-dimensional arrays) and there are computations that depend on the accessed data. The hardware prefetcher is ineffective in this case, because there is no predictable pattern. Helper threads, on the other hand, prefetch correctly and stay ahead of the main thread. Our experience shows that most memory-intensive loops meet both criteria and it is possible to extract more than $2\times$ speedup in some cases by using one helper thread for this type of code. However, ICP can still be useful even for regular access patterns that hardware prefetchers excel at, for three reasons. First, the data ICP prefetches is brought into a separate cache, which does not conflict with the main thread's cache. Second, the prefetching puts no pressure on the main thread's memory interface (e.g., miss status holding registers). Finally, ICP's software prefetch threads can fetch across page boundaries (unlike most hardware prefetchers) and the amount of incorrect prefetching is minimal.

The scalability of ICP is very different than traditional parallelism, another reason that the two techniques are synergistic. While in the best case traditional parallelism scales slowly but steadily (e.g., linear speedups), ICP tends to scale quickly then tail off. If a single helper thread (per main thread) can complete its chunk faster than the main thread, one helper thread suffices and there is little or no gain to adding more. But with that single helper thread, it can often get performance that significantly exceeds linear gains. However, if the ratio of computation to memory access is low, we may need multiple helper threads operating in parallel to fully cover the main thread's accesses. In that case, performance continues to scale, to some degree, as we add helper threads.

Unlike other helper thread prefetchers, ICP does all of its work (either main thread execution or helper thread prefetching) on chunk boundaries, where a "chunk" is a set of iterations expected to access a certain amount of data. Chunking provides two advantages. First, it reduces the cost of synchronization and thread migration by only doing them once for a chunk. Second, it prevents helper threads that run faster than the main thread from running too far ahead and evicting prefetched data from the local cache before the main thread uses it. Chunk size determines the level of cache that ICP targets. Smaller chunks that fit in the L1 cache make prefetching the most effective, but results in frequent synchronization that increases total migration overhead. Chunks that target the L2 cache reduce migration overhead, but provide smaller prefetching gains.

The effectiveness of inter-core prefetching also depends on the amount of data overlap between chunks. Write sharing can result in prefetched data being invalidated before use, and read sharing is often correlated with high cache hit rates (in which case ICP is less effective).

3.2 ICP based coalition threading

Our coalition threading implementation uses ICP as the non-traditional component. For clarity, we use the term *par-ICP* to describe ICP implemented on top of traditional parallelism, and applied to each of the original parallel threads. *Seq-ICP* is ICP applied alone, running on top of a single-thread version of an application. Coalition threading refers to choosing the best technique (seq-ICP, par-ICP, or traditional parallelism) for a particular loop or for all the loops in an application.

Par-ICP applies inter-core prefetching to each original or *main* thread. If an application has N main threads and ICP provides M helper threads per main thread we need a total of $T = (N + N \times M)$ hardware contexts. When N is one, we only exploit helper thread parallelism. Likewise when M is zero, the application is running with just traditional parallelism.

Note that the choice for the value of M is critical for parallel code. In prior work, it was assumed that other cores were otherwise idle, and adding more helper threads was close to free. For parallel code, there is typically a high opportunity cost, because even increasing M by one reduces the number of main threads by a factor of $(2 + M)/(1 + M)$. For example, for a total of 24 threads, we can either have 24 main threads with no helper threads, 12 main threads each with one helper thread, or 8 main threads each with two helper threads. Recall that for ICP, more than one helper thread is only useful when it takes more time to prefetch a chunk than to execute that chunk. Thus, we want our prefetch threads to be as efficient as possible, and to use no more threads than necessary to leave more threads/cores for traditional parallelism. Fortunately, in most cases, ICP achieves nearly all its benefits with M=1.

Several new factors impact the effectiveness of ICP when applied to parallel code, including parallel data sharing and synchronization overhead.

Data sharing (including false sharing) has a stronger negative impact on the effectiveness of par-ICP than it has on seq-ICP. For read sharing, if the main threads share data then so do helper threads, both with each other and with other main threads. Thus, a single cache line can be in many caches. This uses the caches inefficiently and potentially leads to expensive upgrade events when a main thread eventually writes the line. Write sharing is even more critical, because more than one thread can now invalidate the prefetched data a helper thread has accumulated.

Conversely, if access to that shared data is protected by barriers or locks, ICP can be extremely effective. There are several reasons for this. First, synchronization overhead slows down the main thread, making it easier for prefetch threads to keep up. Second, those applications tend to scale poorly, so even if ICP is not helpful at low thread counts, it can be better than traditional parallelism at high thread counts. Third, loops with frequent communication are often vulnerable to load imbalance – in those cases it is often better to make every main thread faster rather than increase the number of main threads.

3.3 Our coalition threading framework

We develop a custom-built source-to-source translator based on Rose [20] to implement coalition threading. Our framework is flexible enough to apply other non-traditional techniques like shared cache prefetching [15] or software data spreading [11]. However, ICP generally outperforms these techniques and so we do not demonstrate them in this paper. The framework supports both OpenMP or pthread code. In the case of OpenMP applications, the framework first converts the OpenMP pragmas into pthread constructs.

Our system decides on a loop-by-loop basis whether and to what extent to apply traditional threading or par-ICP. The framework uses a heuristic to decide the threading technique for each loop and automatically produces code that implements the selected threading technique. The new code generates a *thread group* of $(1 + M)$ threads for each of the original threads.

Threads within a thread group can trade places with one another, but cannot migrate across thread groups. We use a user-space thread migration system to make migration as cheap as possible, as in [12].

The framework uses all $(N + N \times M)$ threads for parallel execution for loops selected for traditional threading. For par-ICP loops, the framework implements ICP within each thread group with one main thread and M helper threads. The framework generates all the code necessary for splitting the loop into chunks and the coordination between main threads and helper threads required for swapping cores within a group. Each main thread provides necessary information (live-ins, chunk id) to the corresponding helper threads to prefetch the right chunk. However, the framework does not automatically create the code that will prefetch data. Instead the framework looks for a callback function associated with a loop that will act as the prefetch function. Generating the callbacks automatically is possible (e.g., as in [13]), but for the applications we study, we construct them by hand.

To maximize prefetching efficiency we introduce two optimizations over the original ICP implementation [12]. First, we use software prefetching instructions instead of regular loads to avoid accidental page faults. Second, we prefetch the chunk in reverse order – iterations that are to be executed last by the main thread are prefetched first by the helper thread. This ensures that the data the main thread will use first will be in the L1 cache even if our prefetch code and chunk size target the L2 cache.

4. METHODOLOGY

This section describes our experimental methodology – the benchmarks that we target and the systems that we use to evaluate coalition threading.

4.1 Evaluation Systems

We use a quad socket, 32-core state of the art AMD Opteron system running Linux 2.6. Table 1 gives more information about the system. We compile all codes using gcc4.1.2 with -O3 optimization level and keep hardware prefetching enabled for all experiments. To measure microarchitectural characteristics like L2 cache misses, we use hardware performance counters.

CPU components	**Parameters**
CPU model	AMD Opteron 6136 2.4 GHz
Number of cores	Quad-socket 32 cores
Level-1 (L1) cache	64-Kbyte, 3 cycles
Level-2 (L2) cache	512-Kbyte, 15 cycles
Level-3 (L3) cache	6-Mbyte, 50 cycles
Cache to cache transfer latency	120-400 cycles
Memory	60-Gbyte DRAM, 150 cycles
Migration cost	$2 - 5\mu s$

Table 1: Memory hierarchy information and migration cost for the AMD Opteron system used in our experiments.

Benchmark Name	**Suite Input used**	**Loops Evaluated**
BT	NAS, B	23
CG	NAS, B	2
LU	NAS, B	14
MG	NAS, C	6
SP	NAS, B	35
Ammp	SpecOMP, Ref	6
Art	SpecOMP, Ref	2
Equake	SpecOMP, Ref	5
Fluidanimate	Parsec, Native	6
Streamcluster	Parsec, Native	2
Graph500	Graph500, 25–5	7

Table 2: Our benchmarks, with the number of loops analyzed.

4.2 Applications

To evaluate coalition threading, we use memory intensive multi-threaded applications from the NAS parallel benchmark suite [2], SpecOMP [1], Parsec [3], and Graph500 [22] workloads. Table 2 summarizes the particular applications and input sizes we use. We did not include SpecOMP benchmarks that were not written in C or C++ code. In addition, we had to exclude a few more benchmarks because Rose could not handle them.

In all, we target 108 loops (Table 2) from 11 applications. All of these 108 loops are programmer-parallelized and they include all the loops that contribute more than 0.5% of the total execution time of the corresponding benchmark when the benchmark executes with one thread.

5. EFFECTIVENESS OF THE DIFFERENT PARALLELIZATION TECHNIQUES

This section measures the effectiveness of coalition threading on a variety of parallel workloads. All the data presented in this section apply for our large 32-core Opteron system.

5.1 Benefits of coalition threading

The usefulness of coalition threading varies on a loop-by-loop basis, so coalition threading for a particular application may include traditional threading for some loops and a combination for others. We will use the term *traditional* when we use only the parallelism as indicated in the parallel programs themselves, seq-ICP means we only apply inter-core prefetching, par-ICP means we

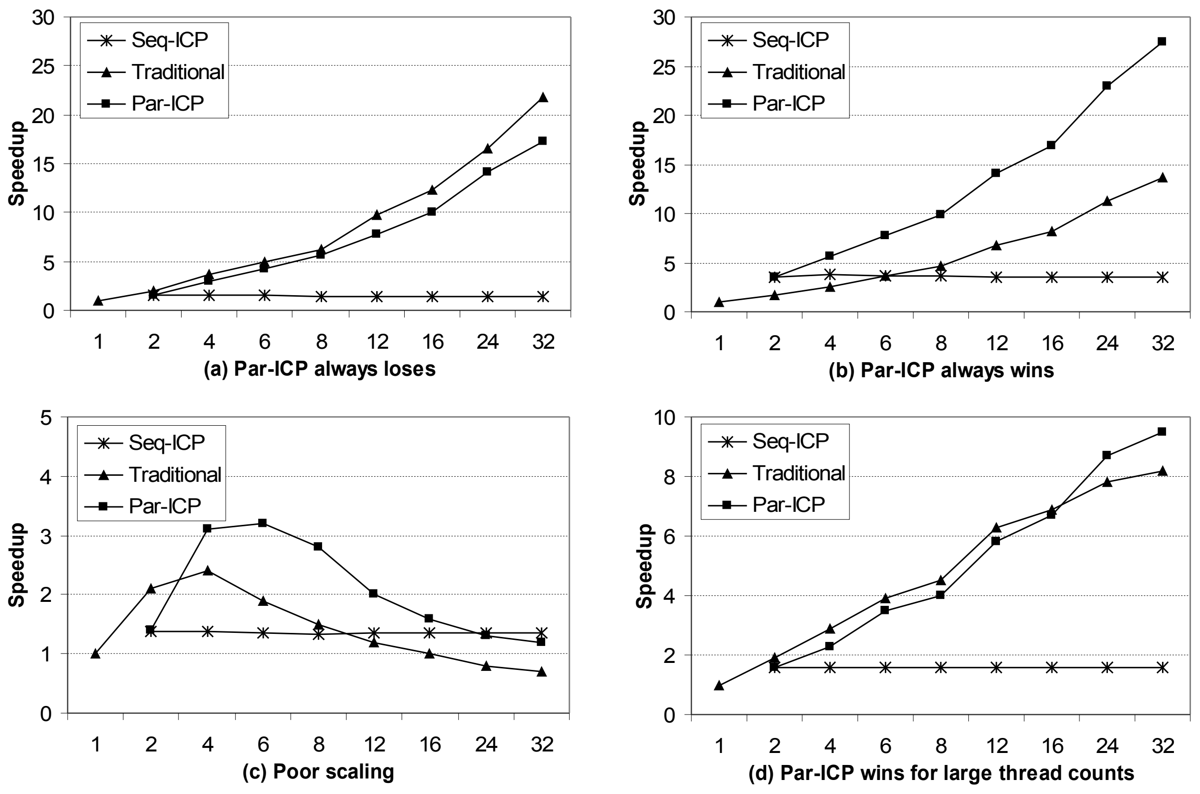

Figure 1: Loops show four different behaviors when we apply seq-ICP, traditional threading, and par-ICP for different thread counts (shown on the x-axis). Most applications contain loops from more than one group. Here, all four loops are from the benchmark *SP*.

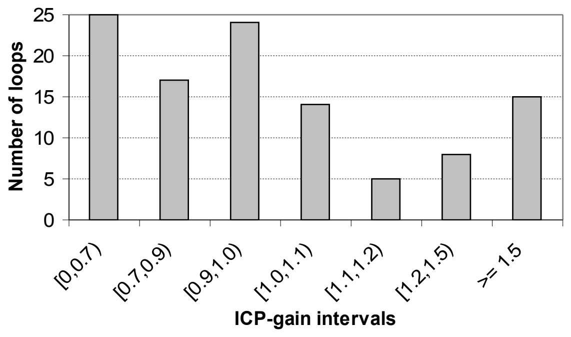

Figure 2: Histogram of 108 loops using $\textit{ICP-gain}_{32}$ for the Opteron system.

combine ICP with the existing parallelism, and coalition threading (CT) means we make an intelligent decision about which type of parallelism to apply to each loop.

We evaluate these techniques on each of the 108 loops in our applications, using between 1 and 32 total threads (and cores). All comparisons throughout the paper apply for the same total number of threads (cores). So, the data point for 32 threads means either 32 main threads (traditional), or 16 main threads with one helper thread (par-ICP), or 1 main thread with 31 helper threads (seq-ICP). We consider other cases of par-ICP (more than one helper thread) in Section 5.2.1.

We see four different behaviors across the loops. Figure 1 compares the performance for four loops that represent these four classes of behaviors. The first class (a) are loops for which traditional threading performs better regardless of thread counts. For loops in class (b), par-ICP consistently outperforms traditional threading. The third class, (c), contains loops that scale poorly (or negatively) even for small thread counts with traditional threading. Behavior for the fourth class, (d), changes as the number of threads increases, despite the fact that these loops tend to have good scalability. In particular, par-ICP loses for small thread counts, but wins for large thread counts. All of these four loops are from one application, *SP*, demonstrating the importance of applying coalition threading on a per-loop basis.

Since all our benchmarks are parallel to start with, ICP alone (seq-ICP) is never the best choice for large core counts. In the more general case, seq-ICP would be an attractive case for many loops that lack parallelism. In our experiments, seq-ICP speeds up almost all of our loops (90 out of 108) over sequential execution, but never over parallel execution; thus we do not consider seq-ICP as an option for the compiler in this paper.

We introduce the term $\textit{ICP-gain}_T$ for a loop to compare performance between traditional parallelization and par-ICP. It is the ratio of the maximum speedup using par-ICP and the maximum speedup using traditional threading while using no more than T threads in total. So, for 32 threads, if a loop gets maximum speedup of $16\times$ using traditional threading and maximum speedup of $20\times$ using par-ICP, the loop has an $\textit{ICP-gain}_{32}$ of 1.25. In the case of coalition threading, we seek to apply par-ICP for only those loops that have $\textit{ICP-gain}_T > 1$.

Figure 2 shows the histogram of the 108 loops to represent the impact of applying par-ICP on each loop in the Opteron system.

The x-axis shows different intervals of *ICP-gain*$_{32}$, and the y-axis shows the number of loops in that category. Applying par-ICP outperforms traditional threading in 42 loops in Opteron. There are 15 loops that gain at least 50% or more, and 7 loops see speedup as high as $2\times$ or more on top of traditional parallelization.

There are a large number (38 out of 108) of loops where the difference between par-ICP and traditional is within $\pm 10\%$. The most common trend we see is for the par-ICP gains to increase with core counts, in fact we observe quite a few cases where the curves appear to be reaching a crossover point just at the limit of cores. Thus, as we continue to scale the number of cores, we expect the number of applications for which coalition threading is profitable to also increase.

Next, we try to understand the factors that cause some of the loops to scale better using par-ICP than traditional threading.

5.2 Understanding loop behavior

We analyze all 108 loops to identify the most common factors that cause the diverse behavior when we evaluate both traditional threading and par-ICP on each loop.

There are some loops where traditional threading always outperforms coalition threading, as shown in Figure 1(a). For these loops, adding more main threads provide better performance than using ICP, even if ICP has something to offer. The 59 loops in this class achieve 19% speedup on average when we apply ICP in the single-threaded case with one helper thread, but if we use that thread for parallel execution, we get, on average, an 83% speedup. These are highly parallel loops with little or no synchronization overhead, few coherence misses, and high instruction level parallelism (ILP) that hides the effect of memory latency, if any.

Conversely, Figure 1(b) represents the set of loops where par-ICP always outperforms traditional threading. This includes both loops that scale well and those that do not. The common factor is that ICP is highly effective in all of these cases. For the 16 loops in this category, executing in parallel gives $1.2\times$ speedup on average for two threads ($N = 2, M = 0$), while ICP gives an average speedup of $1.9\times$ using just one main thread and one helper thread ($N = 1, M = 1$). For some loops, we even see speedup as high as $3.5\times$. These loops have a significant number of L2 misses per instruction, and have little memory level parallelism.

In our analysis, we also find several loops that scale poorly for traditional threading (overlaps with the prior category in some cases). These loops stop scaling beyond some thread count or even start scaling negatively, as in Figure 1(c). We observe three main causes for such behavior. First, some loops have limited scalability because there are not enough iterations to distribute over all threads. This creates load imbalance and does not amortize the cost of parallelism with only one iteration per thread. Second, there are some loops where computation increases significantly (linearly in some cases) with the number of threads. For example, threads can have a private copy of the shared data and can update that copy independently and merge afterward instead of using locks to access that shared data. Finally, some loops execute barriers very frequently and so synchronization time dominates the execution time as the loop is scaled. In all these three cases, par-ICP always works better than traditional parallelism for higher thread counts.

Some loops (total 33) exhibit more complex behavior. Their behavior changes more slowly with the thread counts, as in Figure 1(d). For small thread counts, performance for traditional and par-ICP is similar or par-ICP performs worse. But for larger thread counts, par-ICP starts outperforming traditional threading and the performance gap grows. This is often just a sub case of the previous category, but with lower synchronization overheads that are just starting to be exposed with high thread counts. We see just a couple of counterexamples (2 loops from *LU* benchmark), where the opposite happens – par-ICP wins for smaller thread counts but loses for larger thread counts. These two loops show super linear parallel scalability when enough cores are used to fit the working set in the L2 caches, making prefetching less effective at those core counts.

5.2.1 Multiple helper threads

We also measure performance using more than one helper thread per main thread. Multiple helper threads can be effective when applied to the single-threaded case. In our experiments, the speedup (averaged over all 108 loops) improves to $1.49\times$ from $1.38\times$ when we add one more helper thread ($N = 1, M = 2$ instead of $N = 1, M = 1$). However, in most cases, this improvement cannot compensate the reduced parallelism because of the decrease in the number of main threads. Additionally, more helper threads saturate off-chip bandwidth in some cases and we do not see the same degree of speedup as we scale to higher thread counts.

We found three loops where multiple helper threads provided better performance than using a single helper thread for the same total number of threads (e.g., $N = 8, M = 2$ works better than $N = 16, M = 1$). These loops each had poor scalability as in Figure 1(c). Using more threads for ICP reduced the number of main threads and improved performance. Even in these cases, the gains were small. So, we restrict ourselves to one helper thread per main thread in the remainder of the paper. This also has the benefit of significantly reducing the decision space for the compiler.

6. HEURISTICS TO APPLY COALITION THREADING

Coalition threading is the process of making a decision about when to apply what type of parallelism to each loop. Because we filter our benchmarks for existing parallelism, we focus here on deciding between traditional parallelism and par-ICP. For the more general case, seq-ICP must also be considered.

Predicting when par-ICP is profitable for a loop, then, is essential. Our compiler tool chain profiles (single-threaded) and analyzes each loop to determine how to extract the most performance using the available CPUs. Depending on the loop's characteristics, it will either apply par-ICP or traditional threading. This section describes the heuristics that our compilation framework uses.

We use a standard machine learning technique, linear classification [8], to develop our heuristics. It exploits profile information that we capture from the single-thread execution of the application to classify loops. We describe this next.

6.1 Linear Classification

A linear classifer [8] is a simple and robust binary classifier widely used in machine learning. It uses a hyperplane of n-1 dimension to best separate a set of n-dimensional points to two classes. The linear classifier library like *liblinear* [8] automatically extracts a weight vector from the input training set to represent the hyperplane. For a particular test point, the classifier computes the dot product of the weight vector and the feature vector of the test point and makes a decision based on that. The number of points that the

Name	Description	Name	Description
PO	Parallelization overhead	BRANCH	No of branches per instruction
CDR	Chunk data reuse	IPC	Instructions per cycle
L2MI	L2 misses per instruction	DRAM	DRAM accesses per instruction
TLBL2IN	L2 misses for TLB walking per instruction	FETCH	Per instruction stalls for not fetching
DTLBIN	TLB misses per instruction	DISP	Per instruction stalls for not dispatching
SSE	No of arithmetic operations per instruction	LSQS	Per instruction stalls for load store queue full

Table 3: List of 12 loop characteristics that we measure for the classification.

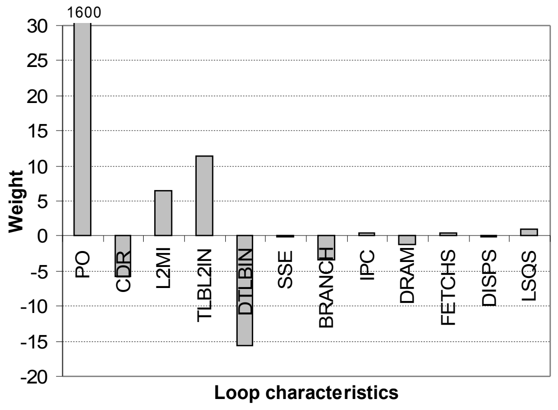

Figure 3: Weight vector generated by the linear classifier that represents correlation.

hyperplane can accurately classify to the correct class determines the accuracy of the classifier.

In our case, we consider each loop as an n-dimensional point where n is the number of loop characteristics that we measure. There are 12 such characteristics, shown in Table 3. These characteristics capture two main things – factors that impact the scalability of traditional parallelism and factors that impact the effectiveness of ICP. Parallelization overhead falls in the first category, while CDR and microarchitectural characteristics (IPC, L2MI, etc.) fall in the second category.

We use hardware performance counters to measure all microarchitectural features. For PO and CDR, we use simple profiling. PO computes the number of barrier or lock executions per instruction while executing with one thread. A high PO implies the loop is unlikely to scale well. We also assign a large PO value to the loops that have limited software parallelism and loops where the computation increases linearly with the number of threads. These special loops also scale poorly with additional threads.

CDR is the percent of data that the main thread chunk shares with the helper thread chunk, when the application runs with one main thread. Write sharing is more expensive than read sharing [12] because of the expensive cache to cache transfers. Thus, we give it more weight and calculate data reuse as: $read\_sharing + k \times write\_sharing$. The value of k depends on how expensive coherence misses are. In our setup, any value of 2 or above works well. CDR varies from 0 to 50% for our loops.

We evaluate our linear classifier in two steps – first we try to find out the features that have strong correlation, and then we measure the effectiveness of the best linear classifier.

6.1.1 Correlation of loop characteristics with the classification

We analyze the weight vector of the linear classifier to understand the correlation of the loop characteristics. The magnitude of the weight defines how strong the correlation is. On the other hand, the sign indicates whether the correlation is positive or negative. In our case, positive correlation indicates a bias toward applying par-ICP.

To analyze the correlation, we construct the linear classifier using all 108 loops and 12 characteristics and compute the weight vector. We do 10-fold cross validation to avoid any bias and take the average weight over all 10 classifiers. For 10-fold cross validation, the data is partitioned into 10 subsets. Each classifier uses a different subset for testing while the remaining 9 subsets are used for training. The cross validation accuracy is the percent of data that are correctly classified. The cross validation accuracy in this case is 79%. There are few parameters (e.g., cost for incorrect classification) which we can tune during the training phase of the classifier. For all of our experiments, we tune these parameters to get the best accuracy.

Figure 3 shows the weight of the characteristics in the classifier. The strong positively correlated characteristics are PO, TLBL2IN, and L2MI, while CDR, DTLBIN, BRANCH exhibit negative correlation. Thus, a loop that has large PO, L2MI, TLBL2IN but low CDR, DTLBIN is most likely to gain from par-ICP. This figure demonstrates several interesting things. First, IPC and different types of stalls (fetch, dispatch) have negligible correlation with the decision process. These are complex metrics and depend on several things like branch misprediction, cache misses, data dependency, resource conflict, etc., but ultimately they do not provide concrete information useful for the classification.

BRANCH has a negative correlation because a high rate of branches indicates irregular code where it is difficult to prefetch the correct data – e.g., they are more likely to contain control-dependent loads or load addresses. L2MI and TLBL2IN have strong positive correlation since ICP primarily converts L2 misses into L2 hits. DRAM and DTLBIN have negative correlation. This is a little surprising since we expect all types of misses to have positive correlation. This is impacted by some cases where the loops are very memory intensive and the data sets are very large, requiring multiple helper threads for ICP to be effective – threads which are better used for traditional parallelism in those cases.

As expected, PO limits the scalability of traditional parallelization, and has strong positive correlation. Similarly, CDR reduces the effectiveness of ICP and has negative correlation.

Since several of the parameters have little impact on the outcome of the classifier, we can simplify it (and the required profiling) by removing some of the metrics. We remove the five characteris-

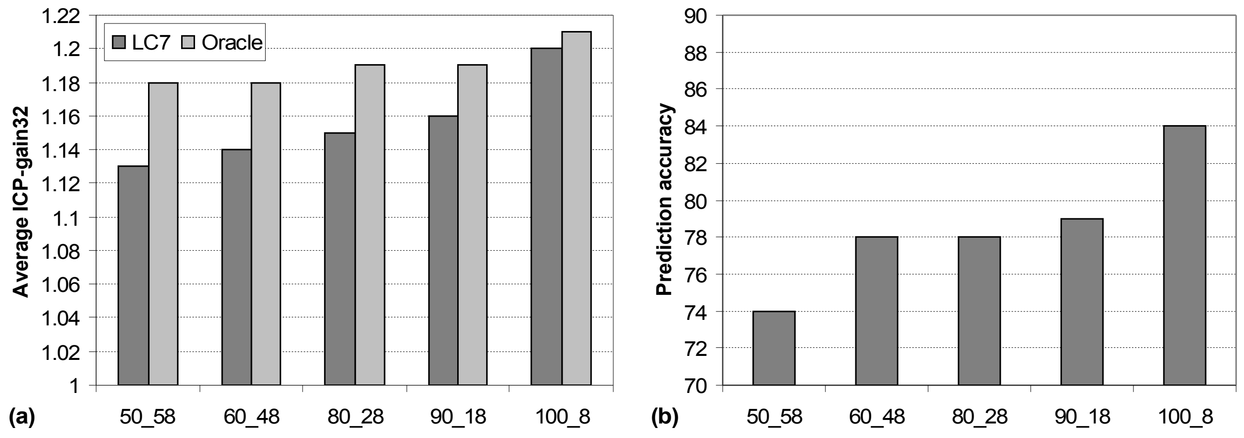

Figure 4: Average $ICP\text{-}gain_{32}$ (a) and prediction accuracy (b) for LC7 using different combinations of training and test set sizes (shown on the x-axis). The first number represents the training set size.

tics (SSE, IPC, FETCH, DISP, and LSQS) that have the smallest correlation and construct the linear classifier with the remaining seven characteristics. The 10-fold cross validation accuracy, in this case, improves to 84%; thus, ignoring the characteristics that are not heavily correlated removes the noise and builds a better linear classifier. On the other hand, skipping some of the important characteristics and picking others does decrease the accuracy, e.g., using PO, CDR, L2MI, SSE and BRANCH decreases the accuracy to 69%. Our best linear classifier uses 7 characteristics – PO, CDR, L2MI, TLBL2IN, DTLBIN, DRAM, and BRANCH. We refer to this classifier as LC7.

6.1.2 Linear classifier performance

This section measures the effectiveness of our best linear classifier, LC7, using standard evaluation techniques for machine learning. We randomly pick *k* loops to train the linear classifier and use the rest of (108-k) loops to test it. While training the classifier, we assume nothing about the test loops and use cross validation for the train loops to avoid bias towards the training set. To cancel out the effect of randomization, for a particular training size of *k*, we repeat the procedure 20 times to pick different sets of training and test loops, and then take the average of the outcomes.

There are two metrics that we measure for the test set – average $ICP\text{-}gain_{32}$ and the number of loops correctly classified in the test set, i.e. prediction accuracy. We compare our results with the oracle classifier which has prediction accuracy of 100%. Figure 4 shows the average $ICP\text{-}gain_{32}$ and prediction accuracy for different values of *k*. Here, the first value indicates the size of the training set while the second one indicates the size of the test set.

We use different training sizes from 50 to 100. The prediction accuracy improves with the train size, because the classifier can capture more information. However, even if we use 50 loops for training and the other 58 loops for testing, we get very good accuracy of 74%. Considering the average $ICP\text{-}gain_{32}$, LC7 loses 5% performance compared to the oracle classifier. For a training set size of 100 loops, the accuracy improves to 84% and LC7 performs just 1.2% less well than the oracle.

We also do another test, where we use all 108 loops to construct the linear classifier and use that to predict the same 108 loops. In this case, the accuracy is 87% and LC7 provides around 1.4% less gain than the oracle classifier. This demonstrates two things – the loops are not linearly classifiable, so we can expect some error, and yet LC7 performs close to the oracle classifier.

For further information, we analyze the loops that LC7 correctly classifies. LC7 can more accurately classify the loops which are highly sensitive to the correct decision – the performance variation between applying par-ICP and traditional is large. If we only consider the loops where par-ICP either wins or loses by at least 5%, LC7 classifies 91% of loops correctly. The accuracy improves to 98% when we consider at least 25% performance variation. This is significant, because the linear classifier does not use any information regarding the possible performance gain or loss. This also explains why LC7 closely follows the oracle in terms of the average $ICP\text{-}gain_{32}$ despite selecting the less effective technique 13% of the time.

Other Decision Heuristics.

In an attempt to simplify the decision process for the compiler, we alternatively take the correlation data learned from the linear classifier, and use that to construct a simple decision matrix. Specifically, we are able to take just four of our highly correlated metrics (PO, CDR, L2MI, and DTLBIN), apply an experimentally derived threshold for each, and classify each loop according to whether it is high or low with respect to these four metrics. By doing so, we are able to replicate the accuracy of the linear classifier. The advantage of this construction is that it uses only four profile measurements and simple table lookup. However, since it does not provide gains above the linear classifier, we do not show the specific results here.

6.2 Application Level Impact

This section analyzes the impact of applying coalition threading (CT) to the applications. There can be several dominant loops in an application, so we have four different schemes to consider – apply seq-ICP to all loops, apply traditional parallelism to all loops, apply par-ICP to all loops and apply the combination of traditional parallelism and par-ICP as suggested by our heuristic. We call the last scheme *heuristic CT* which applies par-ICP to the set of loops chosen by the heuristic and applies traditional threading to the rest. For heuristic CT, our coalition threading framework uses the LC7 heuristic described earlier. We also implement *oracle CT* that ap-

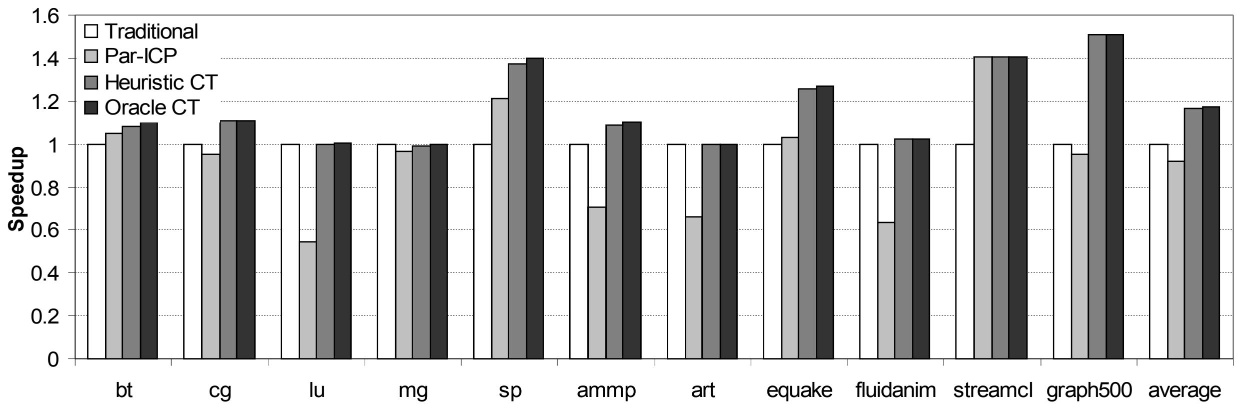

Figure 6: The graph compares performance of oracle CT and heuristic CT with other techniques in Opteron. The improvement is as high as 40%.

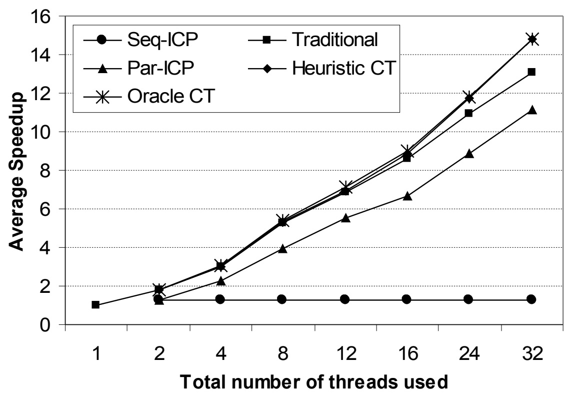

Figure 5: The scalability of the speedup averaged over all benchmarks using different techniques.

plies coalition threading using the oracle heuristic to evaluate the performance of the LC7 heuristic at the application level.

Figure 5 shows the effect of applying all the schemes on our benchmarks in the Opteron system. The y-axis here denotes the average speedup over all benchmarks for a particular number of threads. The seq-ICP scheme cannot compete with other schemes, because most of them scale with traditional threading, whereas seq-ICP scales only marginally beyond 2 cores. Par-ICP scales better than traditional threading and gradually catches up – 28% performance loss for 2 threads vs. 15% loss for 32 threads. The impact of par-ICP is more prominent in case of heuristic CT. For all thread counts, heuristic CT outperforms traditional threading. The improvement is 1% for 4 threads, but reaches 13% for 32 threads. As expected, the utility of coalition threading increases with core count. There is no reason to expect this trend does not continue. Figure 5 also shows the strength of the LC7 heuristic. It performs almost the same as the oracle heuristic.

Next, in Figure 6, we compare the performance of heuristic CT with par-ICP, traditional threading, and oracle CT across all benchmarks. The values on the y-axis are normalized to the best possible speedup by traditional threading using no more than 32 cores.

In the Opteron machine, we see gain as high as 21% for *SP* and 40% for *streamcluster* while using the par-ICP scheme. Oracle CT improves performance further from 21 to 40% for *SP* by filtering the loops that do not get any benefit from par-ICP. However, for *streamcluster*, both the key loops get benefit from par-ICP and oracle CT gives the same speedup as the par-ICP scheme. *Graph500* shows the importance of using a good heuristic. For this application, heuristic CT provides 51% speedup vs. the 5% loss by par-ICP. Overall, the oracle CT gives an average of 17.4% gain across all benchmarks, and an average gain of 21% considering only the nine benchmarks where we apply par-ICP to at least one loop. Compared to that, heuristic CT gives an average of 16.7% gain across all benchmarks and performs very close to the oracle CT. The small number of loops where our heuristic does not make the correct decision have negligible impact on the overall application run-time. So our developed heuristic is nearly as effective as the oracle heuristic.

7. CONCLUSION

This paper describes Coalition Threading, a hybrid threading technique that combines traditional parallelism with non-traditional parallelism, and demonstrates heuristics to find the best combination, which can be used to direct a parallel compiler. We analyze the effectiveness of coalition threading using inter-core prefetching as the non-traditional component and observe that for a number of parallel loops, adding ICP on top of traditional threading outperforms traditional threading alone. Our results show that in a 32-core Opteron system, there are 20 loops out of 108 where coalition threading provides more than 30% improvements on top of the best possible speedup by traditional threading. Coalition threading has a strong impact on application level scalability (up to 51% performance gain in some cases) when we apply the right technique to the right set of loops. Our best heuristic makes the correct decision 87% of the time and can accurately identify 98% of the loops where the correct decision is critical. Heuristic-based coalition threading gives an average of 17% improvements across our benchmarks and is only 0.7% less than what an oracle provides.

Acknowledgments

The authors would like to thank the anonymous reviewers for many useful suggestions. This work was supported by NSF grants CCF-1018356 and CCF-0643880 and by SRC Grant 2086.001.

8. REFERENCES

[1] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Workshop on OpenMP Applications and Tools*, 2001.

[2] D. H. Bailey, E. Barzcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. *IEEE Concurrency*, 1993.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[4] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the International Symposium on Computer Architecture*, 1999.

[6] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precompuation. In *Proceedings of the International Symposium on Microarchitecture*, 2001.

[7] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

[8] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[9] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[10] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2003.

[11] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Software data spreading: leveraging distributed caches to improve single thread performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[12] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[13] D. Kim and D. Yeung. Design and evaluation of compiler algorithm for pre-execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[14] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[15] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the International Symposium on Microarchitecture*, 2005.

[16] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the the International Symposium on Computer Architecture*, 2001.

[17] E. Lusk and A. Chan. Early experiments with the openmp/mpi hybrid programming model. In *Proceedings of the International Conference on OpenMP in a new era of parallelism*, 2008.

[18] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *12th International Conference on Supercomputing*, 1998.

[19] F. Mueller. A library implementation of posix threads under unix. In *In Proceedings of the USENIX Conference*, 1993.

[20] D. J. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *14th Workshop on Languages and Compilers for Parallel Computing*, 2001.

[21] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[22] B. W. B. J. A. A. C. U. G. C. Richard C. Murphy, Kyle B. Wheeler. Introducing the graph 500. May 2010.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[24] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[25] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. *SIGPLAN Not.*, 35(11), 2000.

[26] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[27] W. Zhang, D. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2007.

[28] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the the International Symposium on Computer Architecture*, 2001.