AP+SOMT: Agent-Programming Combined with Self-Organized Multi-Threading

Yves Lhuillier Olivier Temam

LRI, Paris South University & INRIA Futurs France
{lhuillie,temam}@lri.fr

ABSTRACT

In order to scale up processors beyond ILP, we explore the exploitation of coarser-grain parallelism. We advocate that a slightly different programming approach, called agent programming (AP), can unveil a large amount of parallelism, potentially simplify the task of optimizing compilers and empower the architecture with the ability to exploit potential parallelism based on available resources. We show that an SMT, augmented with dynamic steering strategies and thread swapping features, is an appropriate solution for such self-organized architectures; self-organized SMT is called SOMT. Using a set of specially written agent-like programs corresponding to classic algorithms, we show that AP+SOMT exhibit better performance, stability and scalability for a large array of data sets, and makes compiler optimizations less necessary. Finally, we outline that the approach can be progressively adopted as a combination of a hardware add-on and C language extensions, much like multimedia support in current superscalar processors.

1. INTRODUCTION

Beyond translating new hardware resources into improved performance, one of the key debates on processor scalability is the appropriate role of the architecture and the compiler. We have not yet stricken the right balance between architecture and compiler roles for exploiting instruction-level parallelism: both excessively complex superscalar architectures or an excessively demanding compiler role in VLIW or Itanium processors can hurt scalability by complicating and thus slowing down processor development (architecture or compiler). However, because processor scalability now requires to look beyond instruction-level parallelism, this research issue will shift to new grounds. There are two main approaches for scalability: (1) executing faster the sequences of dependent instructions, or (2) exploiting coarser-grain (coarser than instruction-level) parallelism. Examples of the former approach are the Grid Processor Architecture (GPA) [11] which maps dataflow graphs on a grid of ALU operators so they execute faster, Chimaera [26] which proposes a similar concept using FPGAs, and Function collapsing [27] which propose to even collapse these sequences at the circuit-level. Examples of the latter approach are CMPs [12], the Raw architecture [19] and the TRIPS architecture [16] which combines multiple GPAs (and thus both approaches). However, all these techniques rely on the compiler: for the first approach, extracting sequences of dependent instructions seems within reach of static analysis; on the other hand, automatically extracting coarse-grain parallelism in a large range of applications is a very old and only partially successful research topic, which is still limiting the adoption of even small-scale parallel machines today. Unlike ILP, coarser-grain parallelism is difficult to extract automatically, by the compiler, especially in pointer-based application where pointer aliasing hampers dependence analysis.

In this article, we advocate that, provided we strike the right balance between the architecture, compiler and *user* effort, it is possible to unveil coarser-grain parallelism, and then to let the architecture dynamically organize computations, i.e., *self-organize* computations in the spirit of out-of-order superscalar processors, but at a coarser granularity and in a much different way. We will show that the right combination of architecture, compiler and user effort can potentially reduce the compiler effort of optimizing for parallelism and locality; for instance, the architecture can be aware of and dynamically exploit available coarse-grain parallelism, achieving better performance than with static parallelization; a program running on a self-organized architecture can behave almost as if it has been statically optimized for the cache; and overall a self-organized architecture exhibits better scalability than a similar architecture with statically optimized programs. We illustrate this approach using SMTs, where all threads are used here to improve the performance of a *single* process, i.e., a process is parallelized over the different threads. In the long term, we view either a form of CMP+SMT (e.g., the planned dual-core hyperthreaded Pentium 4), or one of the solutions mentioned in the above paragraph, as a more likely path to scalability than single SMTs; but investigating self-organized CMP+SMT or other solutions are beyond the scope of this study.

The key to our approach is to view (and write) programs not as a single large code, but as a set of "agents" interacting together. What we are advocating is to rethink programs, sometimes at the algorithm level, so that they are expressed in an agent-based way; we do not suggest to parallelize existing programs, we believe that it can be an excessively complicated task for many programs. Our experience showed that writing classic algorithms in an agent-like way is fairly intuitive for the user: many operations can be viewed as one or a set of agents "crawling" over a few data structures and performing a set of operations on each data item. By translating her view of the algorithm and data structures in an agent-like program, the programmer implicitly passes information on parallelism and locality without explicitly parallelizing or managing memory. The architecture then takes advantage of this information by *dividing* agents to exploit parallelism, and by slowing/accelerating agents to enforce proper memory behavior. In fact, the roles of both the architecture and the compiler become fairly simple. Finally, it is important to understand that we do not require the user to be aware of the underlying architecture; the additional semantics provided by the user is passed in an effortless manner, unlike architecture-oriented hints for the Itanium, for instance; so *writing* an agent-like program requires little effort, but granted, *re*writing a program is a significant effort, and it will slow adoption.

However, we also designed our approach so that progressive adoption is possible in practice; for that purpose, we propose to augment SMTs with hardware support for agent-like programming, and language extensions that a user may or not take advantage of, much like multimedia SIMD instructions in current superscalar processors. Our base architecture is an SMT, which has assets of its own, such as high server or multi-workload performance [21], and which is about to become more widely adopted, see Intel's Pentium IV and IBM's Power5; moreover, the hardware overhead for supporting agent-based programs is limited. Also, instead of imposing a new language, we simply propose a set of C/C++ language extensions as a method for writing agent-like programs, so that not only the syntax remains almost the same as C/C++, but the user is free to program traditional monolithic codes and not take advantage of the agent programming support.

2. AP: AGENT PROGRAMMING

2.1 Principles and Benefits

The notion of agent programming is tied to the notion of data structures; let us introduce it below through the example of Figure 1, which shows the C and assembly versions of a simple program that increments all values of a binary tree. We first outline how a superscalar architecture would behave, and then how an agent version of the same program running on an SMT would behave.

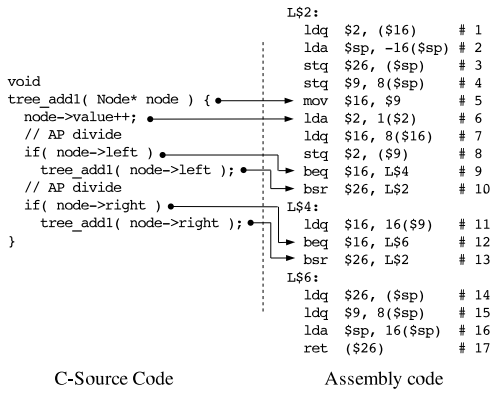

Figure 1: *C and assembly versions of a tree traversal.*

An ILP-oriented architecture like a superscalar (or a VLIW) processor can take advantage of the parallelism among instructions, e.g., instructions 5 and 6 in Figure 2. In a superscalar processor, the loop will be implicitly unrolled within the instruction window, see Figure 3, and the processor can even take advantage of parallelism among instructions from consecutive basic blocks, e.g., instructions $5a$, $6a$, $5b$, and $6b$. Since these consecutive basic blocks actually correspond to operations on consecutive items of the data structure, the processor is executing these operations in a parallel and interleaved way. Note however that due to pointer dereferencing (for accessing the next child node), the instructions corresponding to two consecutive items of the data structure cannot start their execution at the same time; the critical path roughly corresponds to pointer dereferencing, or, in other words, to scanning the data structure in sequential order. Techniques [4, 15] exist to speed up pointer chasing and pointer-based data structure (pre)fetching, but in the end, the data structure is still scanned in a sequential order, one data structure item at a time.

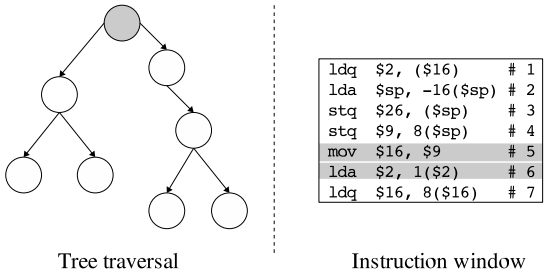

Figure 2: *ILP exploitation.*

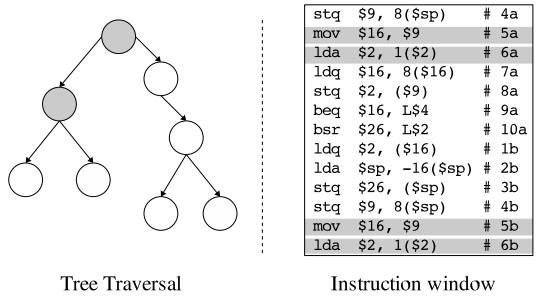

Figure 3: *ILP exploitation across two tree items.*

Now, in this example, any two data structure items can be scanned in parallel, not just two consecutive items; moreover, data structure items need not be scanned in sequential order, one item at a time. Assume we consider the set of operations on a data structure item as a *task*, any two such tasks can be executed in parallel. It is this task-level parallelism that we want to take advantage of, in addition to instruction-level parallelism. Task-level parallelism is coarser than instruction-level parallelism, and similar though slightly finer than the coarse-grain parallelism exploited in SMP machines. However, the key difference is that task-level parallelism is not statically managed as in SMP machines, it is dynamically managed as in superscalar processors.

Coming back to the example of Figure 1, the user now adds the `AP divide` annotations before the recursive calls to let the architecture know it can perform the tasks on each tree item separately, and thus spawn agents at each call (annotations are explained in the next section). This annotation is not a program optimization, it is basically the user passing intuitive information on her representation of the way the program is processing the data structure (in a given order, in any order, etc). For many algorithms, programmers usually have a mental representation of program behavior on data structures which can be passed effortlessly, and which is usually sufficient to characterize when agents can be created. From a compiler analysis point of view, reverse-engineering a pointer-based C/C++ program for dependence analysis is a daunting task; however, from a user point of view, expressing the local properties of an algorithm and its interaction with data structures is far more simple. We are also investigating more elaborate syntax extensions, in the spirit of the STL C++ library [18], where the user can specify data structure processing and pass this information (even more) implicitly.

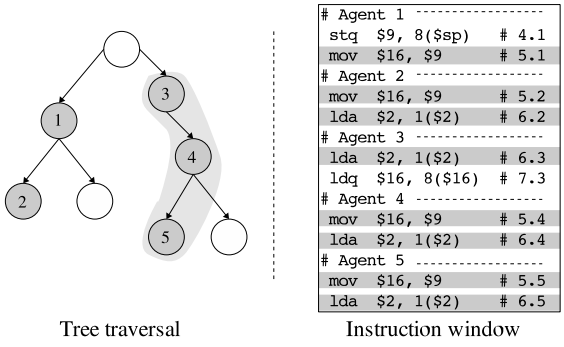

Figure 4: *Agent Programming.*

Once an agent has processed a data structure item, it attempts to process (and thus move to) all *neighbor* data structure items. Assuming the user has imposed no restriction on the order in which data structure items must be scanned in the agent program, the architecture can actually *replicate* the agent instead of executing it in a given or arbitrary order, so that they actually *flood* the data structure, see grey nodes in Figure 4. The architecture controls the number of agents, and implicitly the flooding, based on available hardware resources. This approach has several benefits.

Exploiting parallelism. First, when the user expresses in which order neighbor nodes must be processed, i.e., no specific order, or a specific order, she implicitly expresses parallelism among agents. Then the architecture is free to dynamically exploit this parallelism by allowing agents to replicate.

Exploiting locality. Second, simply enabling the architecture to stall or not agents based on how efficiently they perform on caches induces an efficient exploitation of a cache-based memory hierarchy. The basic principle of caches is that once a data has been brought from memory, it should be heavily exploited before being returned to memory. So, simply privileging agents that perform best on caches is a way to enforce a proper utilization of caches. By making the architecture aware of agent behavior on caches, it can decide to stall the worst performing agents; as a consequence, these agents stop polluting the cache, and best performing agents can use most of the cache space for their own data items.

Consider now the tree scan of Figure 1 is performed again, assume that the cache can only contain one branch of the tree and currently holds the right branch highlighted in Figure 4. Then, the agent moving along that branch will execute faster, and the role of the architecture is to amplify this privilege by slowing down and even swapping out other agents. As a result, other agents do not have time to fetch new data in the cache that would evict present and useful data. Naturally, once the agent of the highlighted branch has completed or is well in the branch, other agents get up to speed again.

Finally, note also that an agent can access (crawl over) several data structures simultaneously, and that several agents corresponding to different code sections can execute simultaneously, they need not be agents working on the same data structures (though it is often the case in standard programs like SpecInt). The practical issues of agent programming are discussed in the next section.

2.2 Syntax

For the moment, agent programming takes the form of a set of annotations that result in assembly-level transformations. These transformations are not yet implemented in a compiler, they are performed by hand, but the corresponding static transformations

AP Annotation	Semantic
`// AP divide`	Divide agent (*before loops and procedure calls*)
`// AP shared`	Atomic access to variable (*before statement using variable*)
`// AP reduction`	Variable for storing the result of a reduction (*before statement using variable*)

Table 1: *Annotations for agent programming.*

are straightforward and can be automated. Most of the transformations either consist in modifying iterators like `for` loops or adding calls to a library of support routines. The main annotations are shown in Table 1.

`AP divide`. This is the main annotation. An agent can divide in two cases: within a loop, and upon a procedure call. The transformations induced by `AP divide` are different in one case and the other.

If the annotation is inserted before a loop, see Figure 5, the dividing transformation consists in determining the loop iterator, its start value, stop value and step, and inserting a call that distributes the upper half of the loop to a new agent, while the current agent finishes the lower half. Loops not amenable to a form where the iterator has an arithmetic progression are ignored. Typically, loops scanning lists cannot be divided for the moment, but we are investigating data structures extensions where lists are represented in a dual array/list mode, facilitating division.

```
// AP divide
for( i=0; i<N; i++ ) {
  a[i]++;
}
```

Figure 5: `AP divide` *applied to a loop.*

If the `AP divide` annotation is inserted just before a procedure call (recursive or not), it can induce division. Consider the example of Figure 1, the agent running procedure `tree_init` reaches the first call (annotated in the source with `AP divide`), it makes a new procedure call and spawns a new agent at the same time, then keeps executing the rest of the procedure.

`AP shared`. This annotation has the same semantic as for parallel machines. `AP shared` specifies variables which multiple agents can update simultaneously. For this latter type of variables, mutual exclusion is implemented through a fast locking mechanism, see Section 3.

`AP reduction`. However, while a support for shared variables is necessary for most programs, it can naturally spoil some of the performance benefits of agent programming. Many operations consist in scanning part of one or several data structures and storing the result in a scalar variable, e.g., cumulating values, finding a min/max,... Such operations can be implemented more efficiently by making local copies of the scalar variable and later synchronizing the final operations on the local copies, provided the operation is commutative and associative. To specify such reduction-like operations, the corresponding scalar variable is simply marked with the above annotation; note that it is implicitly shared. Consider the example of Figure 6, where the tree scan is now used to cumulate the tree values in scalar `a`; multiple agents will be spawned, creating local copies of `a` which will contain the partial sums, and at the end of each agent execution, the local copies are cumulated back to the true (and shared) scalar variable.

```
tree_add( Node* node ) {
  // AP reduction
  a += node->value;
  // AP divide
  if( node->left )
    tree_add( node->left );
  // AP divide
  if( node->right )
    tree_add( node->right );
}
```

Figure 6: *AP reduction.*

3. SOMT: SELF-ORGANIZED SMT

As mentioned in the introduction, our target architecture is SMT. SMT is a natural hardware support for agent programming because a lightweight thread is a simple means for implementing an agent. Other architectures, like CMP, would also provide an adequate support. An SMT must be augmented with three features to support agent programming: (1) thread division, (2) thread activation/deactivation, (3) fast thread synchronization support. Our SMT implementation is similar to the one proposed by Tullsen et al. [22], see Figure 7. There are 8 hardware contexts, 32 registers per context, and the issue width is 16 instructions, using the Icount 4.4 policy [21], i.e., instructions are fetched for 4 threads per cycle, 4 instructions per thread. Each active thread has its own hardware context, which includes the thread state (see below for the different states), the thread registers and the PC.

Thread division/replication. The SMT model already allows multiple threads with separate contexts to be executed in parallel. In the proposed architecture, a thread may, by means of a `New THRead` instruction (`nthr`), divide itself into two new threads; `nthr` is inserted wherever the `AP divide` annotation is used, e.g., see Figure 1. The architecture is free to ignore the instruction and not perform a thread division if available hardware resources don't allow it.

The `nthr` instruction performs the following actions. The instruction is initially treated as an unconditional branch. Upon execution, the instruction creates a new thread by seizing a hardware context. A hardware thread context can have three states: `free` (not allocated to an agent/thread), `active` (instructions are fetched), `stall` (instructions are not fetched). After the `nthr` is decoded, a free destination hardware context is chosen, and this chosen context switches from state `free` to state `stall`.

When instruction `nthr` retires, all thread registers to which instruction `nthr` belongs are copied into the registers of the new hardware context, the PC is set to the first target instruction, the hardware context of the destination thread transits to `active`, and agent instructions are fetched. The thread registers are only copied at the commit stage because `nthr` could be speculative; it would also be possible (faster but more costly) to speculatively copy register map tables. If the `nthr` instruction is on the wrong path and a thread has been started, the corresponding context transits back to `free` state. Note that fetching agent instructions following `nthr` is delayed by the pipeline length. However, we also found that delaying the thread start time had limited impact on performance due to the large amount of parallelism and overlap among threads/agents in most cases, so this optimization did not seem worth the added hardware complexity.

Every agent ends with a `Kill THRead` instruction (`kthr`). Upon decoding `kthr`, the corresponding thread transits from the `active` to `stall`, and stops fetching instructions. When `kthr` reaches the commit stage, if the thread is not the last living one, the thread is killed and the hardware context is deallocated. Otherwise, if the thread is the last one, it is *re*activated to execute the remainder of the program.

Thread activation/deactivation. The proposed model also relies on the ability to swap in and out threads in order to have more threads (agents) than hardware contexts, much like a superscalar processor has more in-flight instructions than the number of functional units.

The architecture handles swaps using a LIFO stack of hardware contexts connected to the register bank, see Figure 7. This solution slows down swapping compared to the 14-cycle register bank selection proposed by Agarwal et al. [1], but does not modify the critical path to the register bank. We estimated swapping, i.e., mainly copying registers, at 200 cycles for 63. We experimentally found that a LIFO stack of 16 entries was sufficient for an architecture with 8 hardware contexts. For 63 registers (31 FP, 31 Integer, 1 PC), the 16-entry LIFO stack has a size of 4kB. No stack overflow occurred in our experiments, but a full architecture should include a system trap for dumping the oldest threads to memory in order to free stack space. When a thread is swapped out to the stack, the hardware context transits back to `stall`, and once the last thread instruction retires, registers are copied to the inactive context stack, freeing the hardware context.

Division strategy. As mentioned before, the architecture decides whether to act upon a `nthr` instruction. The strategy is greedy unless threads are dying quickly, meaning the parallel sections are too short with respect to thread creation overhead. Precisely, an `nthr` instruction is executed if there is a free hardware context, and if the number of threads which died in the past N cycles is smaller than *NumberHardwareContexts/2* ($N = 128$ in our experiments).

Scheduling and swapping strategy. The scheduling policy of SOMT is ICount.4.4 [21], i.e., a policy that privileges best performing threads, which are more likely to efficiently use functional units. In addition, we have implemented a *swapping* strategy to evict threads incurring long delays, mainly due to long memory latencies, much like in large-scale multi-threaded machines [2]. As a result, it is solely based on the observation of the threads cache behavior.

Each load latency is compared against the average latency of the last 1000 loads; if the latency is higher, a thread counter is incremented, otherwise, it is decremented; when the thread counter crosses a threshold (256 for an initial value of 0), the thread is swapped out if there is no free hardware context (i.e. contexts are used at full capacity).

Fast thread synchronization techniques. As proposed in [23], mutual exclusion for accessing shared variables is implemented using a fast locks table, see Figure 7. The lock is set by a `Memory LoCK` instruction (`mlck`) on a given address. The lock is set on the base address of the shared object to be accessed, independently of the object size. If another `mlck` instruction wants to access a locked address, the following instructions are squashed, the thread transits to the `stall` state and the thread id is stored in the *Locking table*, see Figure 7. Each entry of the table has three fields, the address locked, an identifier of the thread possessing the lock, and an identifier of the oldest thread stalled by the locking thread. Thus when the locking thread releases the lock, the oldest waiting thread becomes the new owner.

4. METHODOLOGY

Simulator. Our SMT simulator is built on top of SimpleScalar version 3.0. We ran experiments on a SOMT processor, an SMT processor using the parameters in Table 2, and an aggressive superscalar processor.

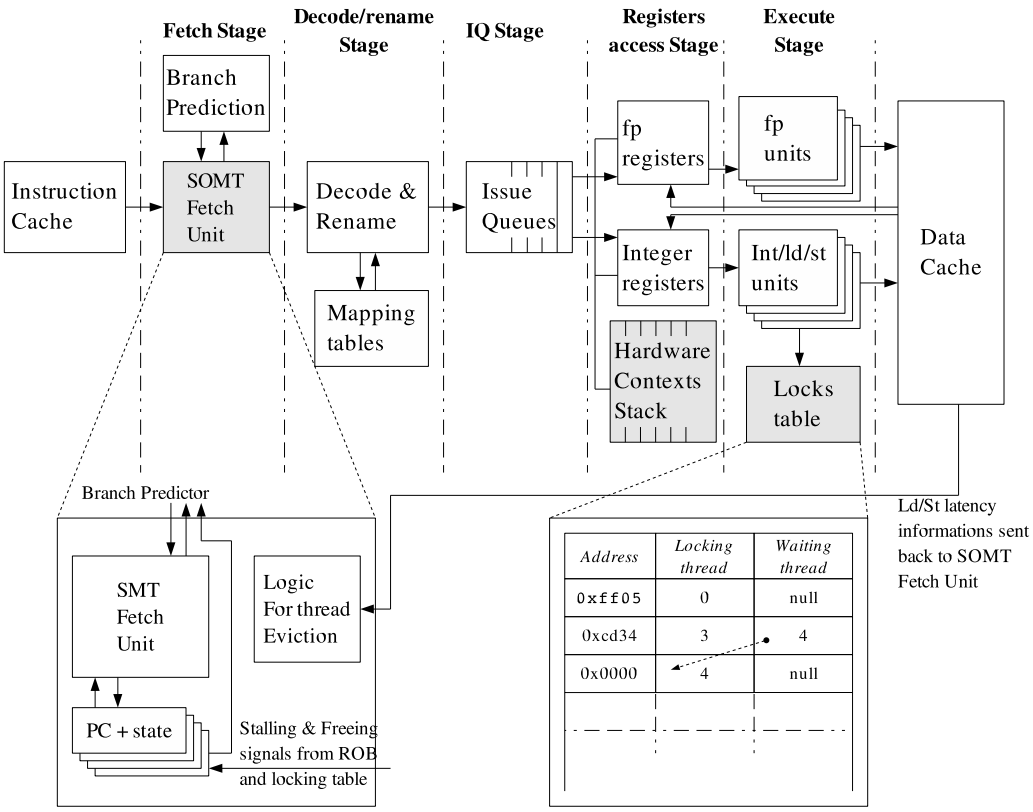

Figure 7: *Self-Organized Multi-Threading.*

Fetch width	16
Issue / Decode / Commit width	8
RUU size (Inst. window - ROB)	256
LSQ size	128
FUs	8 IALU, 4 IMULT, 4 FPALU, 4 FPMULT
Branch	Combined, 4K entries bimodal, and 2 level Gap predictor, 8K 2nd level entries, 14 history wide , 1K meta-table size 7 cycle BR resolution
Memory Latency	200 cycles
L1 DCache	8kB, 1 cycle
L1 ICache	16kB, 1 cycle
L2 Unified Cache	1MB, 12 cycles

Table 2: ***Baseline configuration of SMT and superscalar processors.***

Benchmarks. Because our approach requires to write programs differently, in an "agent way", to take advantage of self-organization, we did not use the Spec programs, but wrote 5 small programs which correspond to frequently used algorithms (and some perform tasks similar to certain SpecInt programs). Our benchmark *LZW* is a compression program which uses the same core algorithm as SpecInt `164.gzip`, *Dijkstra* is a shortest path algorithm used in network routing, *Perceptron* emulates a perceptron (as the SpecInt `179.art`), *QuickSort* is an implementation of the sorting algorithm, and *MxV*, *MxM* are the classic matrix-vector and matrix-matrix multiply. The execution times of these different programs vary from several hundred thousand cycles to several hundred million cycles depending on data set size. For each program, we used multiple data sets (from 10 for *Perceptron*, to one thousand for *QuickSort*) in order to ascertain the stability of the agent version of each program over a sufficiently large range of data sets. For each program, we derived a standard imperative version to be run on a superscalar processor, a statically parallelized version (see below), and our agent version to be run on SOMT. All benchmarks were compiled on an Alpha 21264 using `cc -O3`.

Static parallelization. While there is a broad literature on coarse-grain parallelization, and parallel versions for several algorithms (*MxM*, *QuickSort*, *Perceptron*, *MxV*) which could bring better performance than our fine-grain thread-level parallelization, we could not find parallel versions of some algorithms (*LZW*[1], *Dijkstra* [2], *MCF*). In order to have a statically parallelized version of all pro-

[1]Derivatives of *LZW* have been parallelized but not *LZW* itself.
[2]The *Dijkstra* parallel versions are intended for very coarse parallelism, and would not perform efficiently on small data sets.

grams, we have derived a static parallel version for each program from our agent version using profile-based techniques. The general principle is akin to iterative parallelization: we run the agent version, monitor how data structures are implicitly being divided by agents, and whenever the number of agents reaches the maximum number of hardware contexts, we record how the data is distributed among agents, and use this distribution as a static task parallelization; we explain this approach for each program in more details in Section 5. Therefore, the comparison of our agent against our static versions is both optimistic and pessimistic. It is optimistic for two reasons: it assumes a static compiler will always be capable of identifying a parallel version of the algorithm, which is not the case, especially for pointer-based applications, and it assumes the compiler will be capable of finding enough parallelism to use all hardware threads available. The comparison is pessimistic because a tuned coarse-grain parallel version of some of these programs exists.

5. PERFORMANCE OF AGENT PROGRAMMING COMBINED WITH SELF-ORGANIZED SMT

In this section, we study the performance of agent-like programs running on an SMT with the appropriate support for replicating, scheduling and swapping threads. As mentioned in Section 4, in most experiments, we compare superscalar execution (i.e., "sequential execution", in the sense that there is no thread-level parallelism) with a statically parallelized program running on a standard SMT, and with an agent version dynamically parallelized on a self-organized SMT (AP+SOMT).

5.1 Dynamical parallelization

Irregular data structures and parallelism. Using *QuickSort* and Dijkstra algorithms, we highlight the benefit of agent programming for programs where load balancing among parallel threads is important. In *Dijkstra*, load balancing depends on which fraction of the graph will be explored by each thread. An agent version of *Dijkstra* works slightly differently than the traditional sequential version. Instead of recording at every step the list of all marked nodes and deciding which one corresponds to the smallest distance, each agent attempts to greedily move to all neighbor nodes. If an agent finds an already marked node with a smallest distance than its own, it does not attempt to move to the neighbor nodes of that node and dies. Implicitly, the graph is flooded with agents, see Figure 8; the lack of neighbors, and the availability of hardware resources determine the creation/destruction of threads. Note that each agent accesses two data structures: the graph itself and an array with one entry per node for recording the current shortest path to that node; the access to the array is protected with `AP shared`.

To build a static parallel version of *Dijkstra* to compare against, we have executed the first steps of the algorithms until 8 different graph nodes have been examined; then the 8 threads of the static version will start from these 8 nodes as if the compiler had been able to split the program and the graph in 8 parts; then each thread will sequentially examine all child nodes recursively, stopping on nodes with a smaller distance (where dynamic agents die). Because performance is so much dependent on the nature of the graph, we have randomly generated 50 graphs of 1000 nodes; Figure 9 shows the distribution of execution time in cycles over the 50 graphs for all three architectures. Clearly, the superscalar version version is usually way slower than the agent version. Even though the static parallel SMT version performs significantly better than the superscalar version, the agent SMT version is consistently more efficient

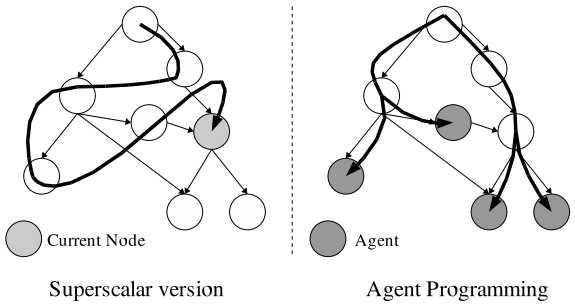

Figure 8: *Flooding the graph with agents.*

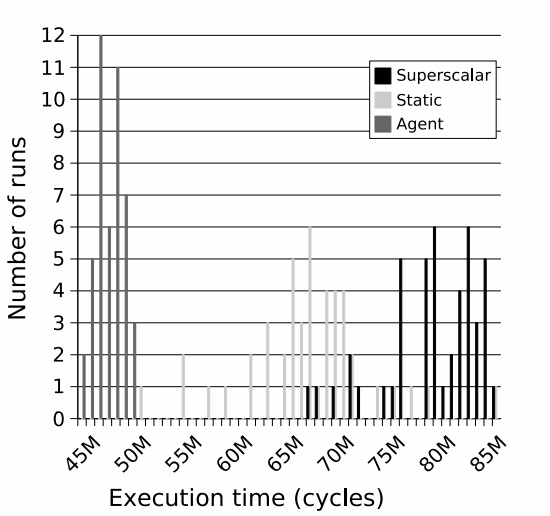

Figure 9: *Distribution of execution time (Dijkstra).*

with more stable performance. The paradox is that the execution of the agent version is non-deterministic, which agent accesses which node first depends on performance issues, so an agent can die sooner in one run than another and their behavior can vary from run to another on the same data set.

In *QuickSort*, load balancing is determined by the relative weight of the two lists obtained after splitting the current list according to the pivot; if one list is large and the other small, one thread will terminate much sooner than the other and hardware resources are not efficiently used. To build our static parallel version of *QuickSort*, we run the first iterations of *QuickSort* until we get 8 lists, one for each hardware context. We record this division, and the static parallel version is started with these 8 individual lists as if the compiler had been able to split the original list in 8 parts. Then each thread executes till completion. Because the *QuickSort* algorithm executes much faster than *Dijkstra*, we were able to test 1000 different lists. As shown in Figure 10, the results are very consistent with *Dijkstra*.

Some existing software parallelization techniques [24] implement load balancing by forking threads, as in our case, except forks are guarded to inhibit forking, e.g., if lists are unbalanced or not large enough. The combination of agent programming and self-organized SMT provides much the same behavior except that forking needs not be guarded since hardware threads are dynamically recycled by the architecture. The main asset of our approach is

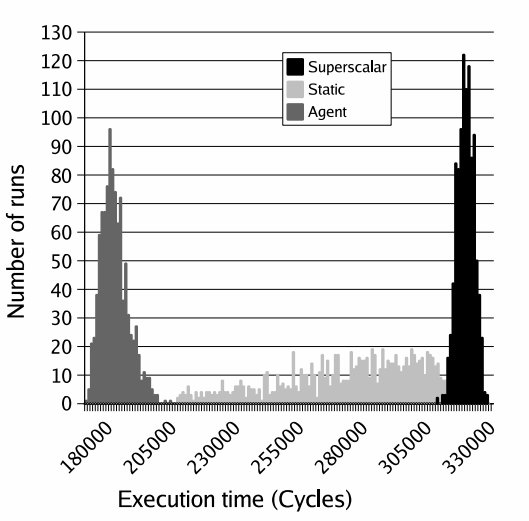

Figure 10: *Distribution of execution time (QuickSort).*

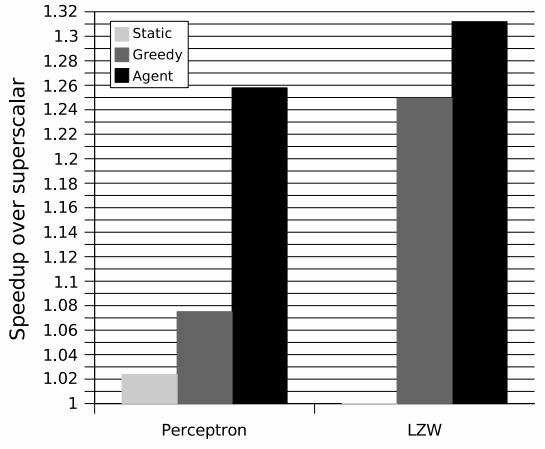

Figure 11: *Speedups of (Perceptron) and (LZW).*

not so much to achieve better performance than existing methods, but to provide systematic and scalable means for extracting performance from a large range of programs.

Small parallel sections. In the two above algorithms, data structures are sufficiently large that it is worth dividing agents as fast as possible. However, we mention in Section 3 that if the life of an agent (a thread) is short, the overhead will eradicate the benefits of exploiting parallelism, and we explained that the architecture monitors the average number of thread destructions per cycle, and takes this metric into account in the cost function driving division decisions. If the number of destructions becomes too large, meaning threads are used to run very short parallel sections, then division is temporarily inhibited. Two algorithms, *LZW* and *Perceptron*, strongly benefit from this cost function.

Figure 11 shows the performance of the superscalar, static and agent version of *Perceptron*, plus the agent version with a greedy (divide always) division cost function. The destruction-aware policy (our default division policy) performs 17% better than the greedy policy. What happens exactly is that the first 7 divisions will occur greedily resulting in 8 threads, then the architecture will start realizing that some of these threads are dying (too) quickly and will inhibit divisions. So, while at least two threads have had an excessively low number of neurons, the remaining threads will not be able to divide as much and create threads (agents) with too few neurons, or only periodically so (until the division is again inhibited).

The static parallel version of *Perceptron* is simply obtained by parallelizing neuron updates in 8 blocks of neurons. The agent version starts with a single agent having all neurons, and it divides by splitting the set of neurons in two as long as the architecture allows. The agent version performs again better than the static version because, as soon as a thread has completed, the hardware context can be reused for another agent; in that case, a currently running agent can divide in two, finishing faster. Because the number of neurons is small (10000 in our example, the same as for 179.art from the SpecInt suite), agents complete very quickly and are reused immediately for splitting the work of other agents. If the number of neurons were very large, the speedup would be smaller because this phenomenon would only show in the final stage of the computations (when agents have few remaining neurons to compute).

The behavior of the greedy versus the destruction-aware division policy is the same for *Perceptron* and *LZW*, see Figure 11. Note, however, that we were not able to obtain an efficient static parallel version for *LZW*, and we found none in the literature, probably because parallelization can only occur at a too fine granularity for SMPs. Our agent version is as follows. The algorithm picks a sequence of characters in a string and wants to match this sequence with the N characters before ($N = 4096$ in our case). The agent can divide the set of N characters, e.g., two sets of $N/2$ characters after the first division, but the agent receiving the first $N/2$ set is also authorized to continue on the second $N/2$ set if it found a matching pattern. A static division brings no performance benefit, but the agent version performs better because threads are used again as soon as they complete, much like in the *Perceptron*.

5.2 Dynamic exploitation of locality

The architecture can influence the execution of agents in two ways: by dividing agents, and by swapping in/out agents, i.e., privileging certain agents over others. In the previous section, we have outlined the benefits of the first action, and in the present section we outline the benefit of the second action.

Our SOMT can influence agent scheduling in two ways. Mildly through the Icount scheduling policy which privileges threads with high throughput. And more strongly through our swapping strategy which privileges threads which perform best on caches. As mentioned in Section 3, this policy fits well with the principles of caches: once data is brought in cache, use it as much as possible before evicting it; so once a thread has brought data in cache, it should be able to use it as much as possible, before other threads, which do not yet have all their data in cache, disrupt well performing threads by polluting the cache with new data.

Applying this policy to agents running on an SMT is akin to changing the order of execution of agents so as to keep data in cache as long as it used by some of the running agents. And this is exactly what most locality-oriented program transformations such as tiling, loop interchange, skewing... [13] attempt to do: change the order of iterations, so that iterations which use the same data are grouped together in time, i.e., execute more or less consecutively. We want to show that a combination of agent programming, where an agent corresponds to (a task on) a block of iterations, and SOMT can behave not so differently than a program statically optimized for locality.

An agent version of matrix-vector consists of the original prod-

uct, see Figure 12(a), which can be split either according to matrix rows or matrix columns. When there are multiple agent division possibilities, we usually enumerate them and let the agent alternatively choose one or the other in a round-robin manner. Note that splitting according to matrix rows favors locality since it tiles the vector in smaller blocks, but having too many (small) blocks can be detrimental as it degrades the locality of the result array C. The execution of the agent version of the original matrix-vector multiply on a SOMT is shown in Figure 13. The figure shows in black the part of the matrix which has been fully used (all computations involving this data have completed), in grey the parts of the matrix which are being used (on-going computations), and in white the parts where computations have not yet started; the different figures show the product at different stages of execution. The initial agent has divided into 8 agents, corresponding to two row-wise divisions and one column-wise division. Note that the architecture privileges the agents using the same matrix columns. What is happening is that division occurred very quickly and the threads working on the left and right columns (left and right block of the vector) both attempted to fetch their vector block at the same time. Because necessarily one of the threads started to have some vector block elements before the other (the left threads), other threads accessing these elements started to get privileged, which in turn, further favored the thread fetching the remaining left vector block elements, and in the end, several of the right threads were swapped out. After the left threads have completed, the right threads gain up speed and perform their part of the computation. The upper half of figure 13 shows the behavior of the statically optimized version of *MxV* (tiled and parallelized), see Figure 12(b), while the lower half shows the behavior of the agent version, see Figure 12(a). One can see that the patterns in the static and dynamic versions are fairly similar, thanks to the memory-aware scheduling strategy of SOMT.

```
void matvec_multiply( int size,
  double* C,double* A,double* B )
{
  // AP divide
  for( int j = 0; j < size; j++ ) {
    // AP divide
    for( int i = 0; i < size; i++ ) {
      C[j] += A[j*size+i]*B[i];
    }
  }
}
```

(a)

```
void matvec_multiply( int size,
  double* C,double* A,double* B )
{
  for( int ii = 0; ii < size; ii+=B ) {
    for( int j = 0; j < size; j++ ) {
      // loop below is parallelized
      for( int i = ii; i < i + B; i++ ) {
        C[j] += A[j*size+i]*B[i];
      }
    }
  }
}
```

(b)

Figure 12: *(a) MxV and (b) tiled MxV.*

We observed, that the self-organization model achieved 61% of the miss reduction obtained by a statically tiled version, and that it achieves 90% of the speedup, over a range of different matrix dimensions.

Dynamic steering has other benefits. Namely, it is more robust than static optimization because it can adjust to unpredictable (or hard to predict) performance variations. Consider matrix-multiply which epitomizes cache conflict issues [9]. A well-known phenomenon is that when the matrix dimension modulo the cache size is small, matrix-multiply performs poorly because cache conflicts strongly hamper self reuse. Such conflicts may be hard to predict [20] and have fueled considerable compiler research efforts in the past ten years [3, 6]. In the context of agent programming, a conflict means the corresponding thread will be performing poorly; with the memory-aware scheduling policy described in Section 3, the thread will be swapped out, and other agents can take advantage of available hardware resources. As a result, the impact of such conflicts on overall program performance is partially hidden; Figure 14 outlines the stability of the agent version of *MxM* compared to the static and superscalar versions, for a 8kB direct-mapped data cache.

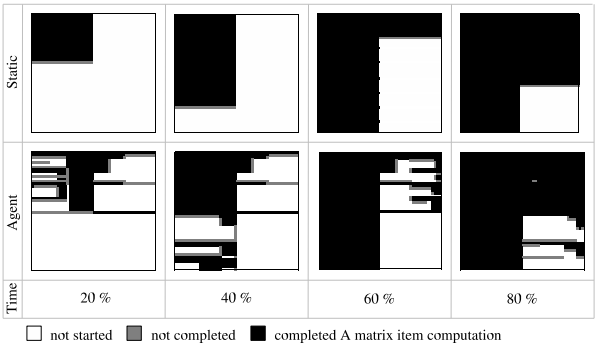

Figure 13: *Tiling-like behavior with MxV.*

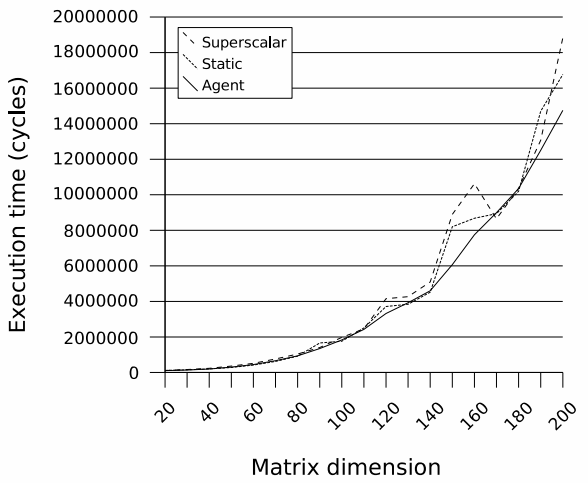

Figure 14: *Hiding cache conflicts with MxM.*

5.3 Scalability

One of the key benefits of combining agent programming with SOMT is scalability. Agent programming provides a large amount of potential parallelism that can be exploited *without* any further program modification when the SOMT is scaled up and hardware resources increase. In Figure 15, we have increased the number of hardware thread contexts T and the number of ALUs ($T/2$ integer ALUs) to evaluate scalability; agent programming and SOMT is compared against the statically parallelized versions of the different programs (note that the number of static threads is equal to $T/2$, i.e., the static version is scaled up for each new hardware di-

mension), and against a very aggressive superscalar processor (instruction window size is 256, and the number of functional units is the same as in the SMTs). We observe that agent programming and SOMT can take advantage of additional hardware resources, and do it more efficiently than statically parallelized versions for the reasons outlined in previous sections, see *QuickSort* and *MxV* in Figure 16 for instance; both the static and dynamic SMT largely outperform the superscalar processor in terms of scalability. A more subtle difference between static parallelization + SMT and agent programming + SOMT is that the latter can also adjust the number of threads (agents) to the amount of work. With a large number of threads, statically parallelized versions can have too small parallel regions which degrades performance (too much overhead), while the SOMT version limits the number of threads because it has detected fast thread destruction, see Section 5.1.

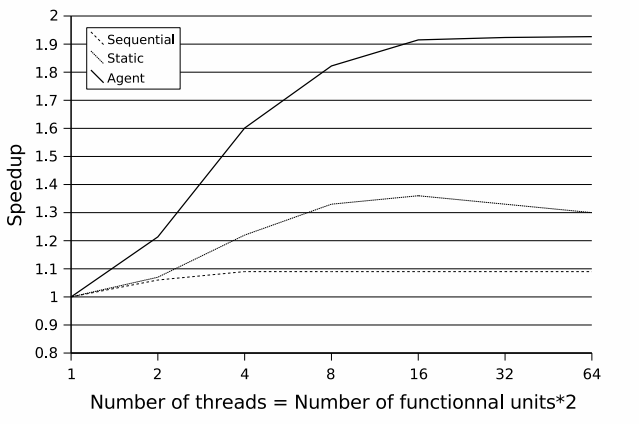

Figure 15: *Scalability (averaged over all benchmarks).*

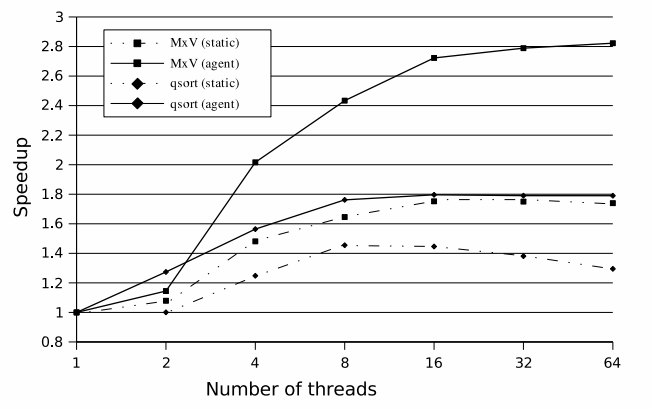

Figure 16: *Scalability (QuickSort and MxV).*

5.4 Example of a larger application.

The best way to take advantage of Agent Programming is to write a full program using this programming model, but it is also possible to apply it (though less thoroughly) to existing applications. As an example, we have annotated *MCF* which spends 45% of its execution time in procedure `refresh_potential`. We have modified this key routine, especially the loop shown in Figure 17. The main loop of the procedure performs a depth-first tree traversal, but the algorithm is such that this order is not compulsory. To avoid the extensive use of recursive calls, the programmer used a double-linked tree, in order to backtrack more efficiently to alternative tree paths (other children) after reaching a branch leaf, see `node->sibling` in Figure 17: the first part of the code from line 2 to line 11 browses the tree from child to child until a leaf is encountered, then the second part of the code backtracks (using `child->pred`) until a sibling is found. In the agent version, division occurs whenever there is more than one child, thus performing simultaneously a depth-first and breadth-first traversing. Applying agent programming on this code section and running it on an 8-context SOMT improves its performance by a factor of 2.14 over a comparable superscalar processor, i.e., a 1.32 speedup for the global application; the test has been conducted using the `train` data set (9 billion instructions were executed).

```
1  while( node != root ) {
       // AP divide
2      while( node ) {
3          if( node->orientation == UP )
4              node->potential =
                 node->pred->potential +
                 node->basic_arc->cost;
5          else {
6              node->potential =
                 node->pred->potential -
                 node->basic_arc->cost;
               // AP reduction
7              checksum++;
8            }
9          tmp = node;
10         node = node->child;
11       }
12     node = tmp;
13     while( node->pred ) {
14         tmp = node->sibling;
15         if( tmp ) {
16             node = tmp;
17             break;
18           }
19         else
20           node = node->pred;
21       }
22   }
```

Figure 17: *Original MCF code with AP annotations.*

6. RELATED WORK

There are other research works on using SMTs to speedup single processes by taking advantage of available threads. Tullsen et al. [10] have investigated this approach using static parallelization and have shown that SMT is able to exploit both ILP and TLP and achieve a 2.68 speedup over a 2-core multi-processor. Another approach for using multiple threads to speedup single processes are *helper threads* [14] which spawn a reduced version of the main process, capable of running of ahead of it, in order to warm up the cache [17] or to provide a feedback on branch behavior [28].

Dataflow-oriented processors like GPA, or RAW [11, 19] aim at removing the limitations of superscalar processors by taking advantage of on-chip space. They do represent an interesting alternative, especially since they alleviate some of the addressing limitations of original dataflow architectures, but their scope (and thus potential parallelism) remains restricted to fairly large but consecutive sets of instructions. Several other architectures proposed speculative execution windows [16] and also increase the potential parallelism by exploring several instruction windows, but the limitations remain almost the same.

Coarser grain parallelism is naturally exploited in classic SMPs, but also in CMPs [12] using static parallelization. Few researchers

have yet explored combinations of dynamic parallelization [25] and CMPs, but they may also represent an interesting alternative to AP+SOMT for extracting and exploiting coarse-grain parallelism.

Finally, there is a large array of research works on dataflow and parallel languages [24, 7] to which agent programming borrows, as well as the notion of Internet agent [5] itself. The only relationship with the latter notion is only intuitive: Internet agents are comparably large programs that crawl the Internet while agents for SMTs are small pieces of code that crawl data structures.

Agent programming is partially inspired by novel and recently proposed computing models, and particularly Blob computing [8], which proposes to control and coordinate a large set of individual computing objects using exclusively local rules (such as division) embedded in the architecture. Blob computing cannot be translated into a realistic architecture because it assumes the hardware substrate is infinitely large and thus cannot adjust to limited hardware resources. Also, the Blob language is cellular-automata based, where the program is written as a set of state transitions, which requires too much efforts from programmers used to existing C/C++ programs. Finally, the programmer has to explicit all object divisions, which is compatible with a greedy division policy when space is infinite, but not with a real finite-size architecture.

7. CONCLUSIONS AND FUTURE WORK

We have introduced a combination of agent programming and self-organized SMT as an approach to achieve better scalability than current ILP-oriented architectures by exploiting coarser-grain parallelism. On a set of algorithms written as agent programs, we have experimentally outlined the benefits of the approach: better exploitation of dynamic parallelism, especially for complex data structures or small parallel sections, better scalability as the number of hardware resources increases, and behaviors not unlike programs statically optimized for locality and parallelism, but without the corresponding compiler effort.

Besides refinements on the language extensions, we are turning to more likely candidate architectures for long-term scalability than SMT, especially a combination of CMP+SMT, i.e., a set of SMT processors connected in a grid-like manner. Agents are then not contained within an SMT, they can move from one processor to the other. Initial research suggests agent programming is well suited to this spatial layout, even though space naturally brings new issues like migration and addressing.

REFERENCES

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: architecture and performance. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 509–520. ACM Press, 1998.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, pages 1–6. ACM Press, 1990.

[3] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290. ACM Press, 1995.

[4] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 279–290. ACM Press, 2002.

[5] Oren Etzioni and Daniel S. Weld. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert: Intelligent Systems and Their Applications*, 10(4):44–49, 1995.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, pages 317–324. ACM Press, 1997.

[7] J-L. Giavitto and O. Michel. MGS: a rule-based programming language for complex objects and collections. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.

[8] F. Gruau and P. Malbos. The Blob: A basic topological concept for hardware-free distributed computation. In *Unconventional Models of Computation (UMC'02), Kobe, Japan*, pages 151–163, 2002.

[9] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74. ACM Press, 1991.

[10] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354, 1997.

[11] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 40–51. IEEE Computer Society, 2001.

[12] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11. ACM Press, 1996.

[13] D. Parello, O. Temam, and J-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance matrix-multiply revisited. In *Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page 31. IEEE Computer Society, 2002.

[14] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 37–48. IEEE Computer Society, 2001.

[15] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 111–121. IEEE Computer Society, 1999.

[16] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

[17] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 171–182. IEEE Computer Society, 2002.

[18] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Hewlett-Packard Laboratories, 1994.

[19] Michael Taylor, Walter Lee, Jason Miller, David Wentzlaff, Benjamin Greenwald, Volker Strumpen, Nathan Shnidman, Ian Bratt, Henry Hoffmann, Jason Kim, Arvind Saraf James Psota, Jonathan, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st annual international symposium on Computer architecture*. ACM Press, 2004.

[20] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 261–271. ACM Press, 1994.

[21] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 191–202. ACM Press,

1996.

[22] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403. ACM Press, 1995.

[23] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*, page 54. IEEE Computer Society, 1999.

[24] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Pursuing laziness for efficient implementation of modern multithreaded languages. In *Fourth International Symposium on High Performance Computing*, pages 174–188, 2003.

[25] L. Rauchwerger Y. Zhan and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proceedings of the The Fourth International Symposium on High-Performance Computer Architecture*, page 162. IEEE Computer Society, 1998.

[26] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 225–235. ACM Press, 2000.

[27] Sami Yehia and Olivier Temam. From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation. In *Proceedings of the 31st annual international symposium on Computer architecture*. ACM Press, 2004.

[28] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13. ACM Press, 2001.