CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs

Pierre Palatin † ⋆
INRIA Futurs, France
pierre.palatin@inria.fr

Yves Lhuillier ‡
LRI, University of Paris Sud, France
yves.lhuillier@lri.fr

Olivier Temam ⋆
INRIA Futurs, France
olivier.temam@inria.fr

Abstract

Since processor performance scalability will now mostly be achieved through thread-level parallelism, there is a strong incentive to parallelize a broad range of applications, including those with complex control flow and data structures. And writing parallel programs is a notoriously difficult task. Beyond processor performance, the architect can help by facilitating the task of the programmer, especially by simplifying the model exposed to the programmer.

In this article, among the many issues associated with writing parallel programs, we focus on finding the appropriate parallelism granularity, and efficiently mapping tasks with complex control and data flow to threads. We propose to relieve the user and compiler of both tasks by delegating the parallelization decision to the architecture at run-time, through a combination of hardware and software support and a tight dialogue between both.

For the software support, we leverage an increasingly popular approach in software engineering, called component-based programming; the component contract assumes tight encapsulation of code and data for easy manipulation. Previous research works have shown that it is possible to augment components with the ability to split/spawn*, providing a simple and fitting approach for programming parallel applications with complex control and data structures. However, such environments still require the programmer to determine the appropriate granularity of parallelism, and spawning incurs significant overheads due to software run-time system management.*

For that purpose, we provide an environment with the ability to spawn conditionally *depending on available hardware resources, and we delegate spawning decisions and actions to the architecture. This conditional spawning is implemented through frequent hardware resource probing by the program. This, in turn, enables rapid adaptation to varying workload conditions, data sets and hardware resources. Furthermore, thanks to appropriate combined hardware and compiler support, the probing has no significant overhead on program performance.*

We demonstrate this approach on an 8-context SMT, several non-trivial algorithms and re-engineered SPEC CINT2000 benchmarks, written using component syntax processed by our toolchain. We achieve speedups ranging from 1.1 to 3.0 on our test suite.

† *Supported by grant from CNRS and LRI, University of Paris Sud, France.*

‡ *Now at CEA Saclay, France*

⋆ *Members of HiPEAC, a European Network of Excellence from the FP6 IST program, DGXIII contract no. IST-004408.*

1. Introduction

There is a consensus in architecture research that processors will now achieve performance scalability in large part through thread-level parallelism, using multi-threaded and multi-core architectures. Therefore, a broad range of programs will now have to rely on and take advantage of thread-level parallelism, including those with complex control flow and complex data structures.

However, there is no consensus on how to easily parallelize such programs, and the issue is now becoming critical. Since automatic compiler-based parallelization has achieved only mixed results on programs with complex control and data structures [13], programs will have to be manually parallelized for the time being. And, writing or debugging parallel programs is a notoriously difficult task [17]. Unfortunately, in the coming years, a broader community of developers will have to be exposed to parallelism because processors will essentially scale through thread-level parallelism.

Since parallel programming complexity comes in large part from the architecture model exposed to the user, architects can greatly help by *simplifying* this model, beyond performance issues. Parallel programming complexity is a multi-facet issue, it includes, but is not restricted to: (1) extracting multiple execution threads, (2) sharing data among threads through locks (and the associated races and deadlock issues) assuming a shared-memory model, (3) finding the appropriate parallelism granularity, (4) mapping tasks to threads. Currently, the user is exposed to all such issues. Automatic parallelization or parallel languages such as HPF [23] relate to the first issue. The currently popular transactional memory model [14] relates more to the second issue. The present article is particularly focused on the third and fourth issues.

While (1) and (2) have received much attention and are indeed important issues, (3) and (4) also play a big role in complicating the task of the user. Deciding whether a code section is worth parallelizing is deeply related to the architecture behavior, and it is difficult to grasp, even for an expert user. Mapping tasks to threads becomes a similarly complex issue for programs with complex run-time control and data flow behavior, which are typically plagued with load balancing issues.

In this article, we propose to relieve the user, and the compiler, of both tasks by delegating the decisions to the architecture, at run-time. Then, for (3), the role of the user

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

is only to indicate if a code section *may* be parallelized, and the role of the architecture is to poll its resources and analyze program behavior to decide at run-time if the code section *can* be parallelized. Furthermore, for (4), the architecture dynamically distributes tasks among threads at run-time. As a result, the user is free to identify parallelism of any granularity without concern for overhead issues, and neither needs to delve into tasks distribution issues.

The aforementioned symbiotic behavior between the user and the architecture can greatly simplify the task of the programmer. In order to implement this modified hardware/software interface, we need to add support in both the architecture and the program.

For the programming support, our proposition elaborates upon component-based programming [34]. Components are not a new language, they are simply the next step in the evolution of encapsulation, beyond procedures and objects. The component contract is to provide tighter encapsulation: total isolation of the component code from the rest of the program (e.g., no shared variable), explicit communications with other components (e.g., input/output ports), so that the programmer can reason only locally on the component code. As a result, this approach is becoming increasingly popular for large development. Early on, Cilk [8] and Charm++ [18] leveraged that approach for parallel programming, because isolation and local programming simplify the task of identifying data dependences in large applications, i.e., issue (2) above. Their granularity of encapsulation is often finer than in traditional component-based environments used for software-engineering purposes.

Cilk and Charm++ showed that components can be particularly well suited for parallelizing programs with complex data and control flow behavior because parallelization can be expressed with a single simple and intuitive action: component *spawning*. The intuition is that a component can encapsulate part of its workload (control and/or data) into a child component and spawn it, e.g., in another thread.

However, both Cilk and Charm++ have two key limitations. First, the user must determine the appropriate parallelism granularity to avoid spawning too small components, which would incur a large overhead. Second, both environments implement component spawning through software run-time systems, which is flexible but also comes with a rather steep cost, adding to the overhead. These two limitations come from the fact that both Cilk and Charm++ were targeting large-scale multi-processors (coarse parallelism) rather than small-scale multi-threaded architectures.

We get rid of both limitations through a better dialogue between program and architecture by: delegating the spawning decision to the architecture, having the program constantly *probe* the architecture for available resources to spawn components, and consequently, taking into account that component spawning can *fail*. Moreover our components do not so much *spawn* child components, as *divide* themselves in half (each taking a part of the control and/or data workload), a bit like the mitosis of biological cells. For instance, a component containing a loop would probe hardware resources at each iteration and split the loop in half whenever a resource is available. This frequent probing allows rapid adaptation in case of complex load balancing (within or across data sets, or even among processes). In addition, we embed adaptive control to monitor the death rate of components (i.e., the rate at which the corresponding threads terminate) and throttle down component division if components die too quickly (too small components).

In addition, we embed adaptive control to monitor the death rate of components and throttle down component division if components die too quickly (too small components).

We have designed our own component framework, and the associated software tool chain, in order to remove most of the overhead of spawning, and especially probing. However, the hardware support for fast spawning can potentially be used with other existing environments, such as Cilk. Such environments could either use the hardware support for conventional systematic division, or keep their division model but adapt their underlying run-time system to leverage the conditional division mechanism.[1] Components are implemented as threads within the architecture. The architecture support, beyond standard thread and lock management, consists of two new additions to the ISA to implement probing and component spawning/destruction, as well as control for steering spawning.

We ported this hardware support to a SMT processor and we demonstrate that the component spawning parallel programming model paired with appropriate hardware support can efficiently execute programs with complex control flow and complex data structures.

One of the main assets of our approach is its ability to dynamically and quickly adjust to varying workloads in small code sections, or fast changing code sections. The approach has other benefits. Programs become more portable and can scale more easily with the number of threads than statically parallelized programs, because they assume nothing on the number of available threads or the nature of the supporting hardware.

Section 2 illustrates the principles of our approach on an example, Section 3 presents the hardware and compiler support for our component approach on a SMT, Section 4 introduces the experimental methodology, Section 5 presents experimental results, and Section 6 discusses related work.

2. Principles and Benefits

In this section, we illustrate the usage and benefits of component-based programming for expressing fine-grain parallelism, and highlight its potential synergy with a hardware support for driving spawning and mapping decisions, which is at the core of our approach.

One must understand that component-based programming is a naturally decentralized form of programming; the program is broken down into separate and independent components. Unlike objects, components do not freely reference other components within their code, they only spawn or pass information to them, in order to ensure encapsulation. Implicitly, component-based programs are well suited to multi-threaded single-core architectures, shared-memory multi-cores, and even distributed-memory multi-cores thanks to their separation properties. In the present article we only target architectures with a single address space. Component encapsulation has two side effects: (1)

[1] Note that the varying cost of probing and spawning could also result in different run-time behavior.

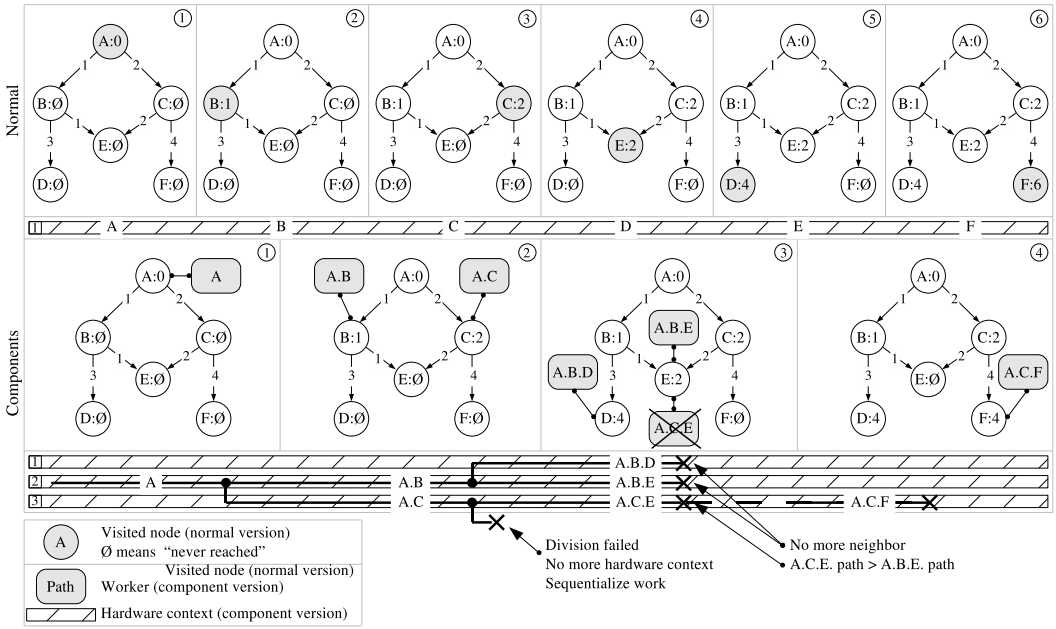

Figure 1. *Progress of normal and component versions of Dijkstra.*

the same specification implemented using components and using plain C or C++ can result in significantly different implementations, if not algorithms; (2) but the component implementation/algorithm is often more amenable to parallelism thanks to a stricter separation of state and decoupled component execution.

As a running example, consider the Dijkstra algorithm, often used in routing [25] to determine the shortest path between a starting node and all other nodes of an oriented graph with weighted edges. The algorithm iteratively walks through the graph, selects a node at each iteration, tags all the neighbor nodes with the path distance from the root node (sum of the path distance so far, and the distance to the neighbor), and finally selects, among all tagged nodes, the node which realizes the shortest path; the algorithm iterates until all target nodes are tagged, as shown in Figure 1 (see `Normal`).[2]

Because tagged nodes are not necessarily neighbor nodes (they could have been tagged during previous iterations), a standard Dijkstra implementation would use a central list to store all tagged nodes. This central list actually hinders parallelization, but it is only an artifact of imperative programming. Obviously, the different paths can be scanned concurrently.

So, a component view of Dijkstra could be the following. A component walks through the graph, recording the traversed path length both in the component and in the traversed nodes.

At any node, if the traversed path length stored in a component is smaller than the path recorded in the node (current shortest path to that node), then a shorter path has been found, and the component records it in the node; otherwise, the component is traversing a sub-optimal path and should thus die, as for `A.C.E` in Figure 1.

[2]The ∅ symbol in Figure 1 denotes an empty set, not 0; it indicates that the node has never been reached.

If a node has multiple child nodes, then the component needs to choose among as many paths. Instead of serially or recursively exploring them, the component can explore all child nodes concurrently by splitting itself into several child components (division), as shown in Figure 1 (see `Component`). The corresponding code is shown in Figure 2(a) (the pre-processed and post-processed versions will be explained in Section 3), a component is called a `worker` in our syntax, and probing+spawning a component is implemented through a `coworker` call. The pre-processor transforms this call into a `switch` statement, see Figure 2(b) which effectively implements the probe+spawn operation. The user writes what happens if the probe fails (usually, that the component simply carries on its serial execution), as shown in the `case -1` statement.

The hardware support takes charge of the remaining decisions. It monitors hardware resource usage, and whenever it sees a spawning request in the form of an assembly `nthr` instruction, it can either decide to act upon it or treat it as a `nop`. For instance, in Figure 1, on step 1, the architecture lets the first component (path `A`) replicate into `A.B` and `A.C` because a hardware context is available. However, on step 2, the three contexts are used, and two components want to replicate, so the architecture denies the replication of one of them (component `A.C`).

Thanks to dynamic steering, hardware resources are not over saturated and fully used when there are enough spawning opportunities. Therefore, a component program paired with hardware support executes more efficiently than the same statically parallelized component program, as shown in Figure 3. This figure shows the execution time of the *Dijkstra* algorithm for 100 randomly generated graphs of 1000 nodes each, on a superscalar processor, a SMT architecture, and a SMT architecture augmented with component support. The x-axis shows the execution time in millions of cycles, the y-axis shows the number of data sets with the same execution time. As can be seen, for almost all data sets, the

0-7695-2732-9/06 $20.00 © 2006 IEEE

```
worker dijkstra {
   //explicit declaration of 'co-workers' for
   //exploring child nodes
   dijkstra  coworker;

   //worker actions
   dijkstra (node_t *node, int distance, node_t *from) {
      // -- worker task on local data structure element --
      mlock(node);
      if (distance >= node->distance) {
         munlock(node);
         delete;   //destroy agent
      }
      node->distance = distance;
      node->parent = from;
      munlock(node);

      // -- worker replication for each child node --
      for (edge_t e = node->edges; e != NULL; e = e->next)
         coworker(e->head, node->distance + e->length, node);
   }
};

//Start
new dijkstra (initial_node, 0, NULL);
```

(a) Source

```
void dijkstra (node_t *node, int distance, node_t *from)
{
   mlock(node);
   if (distance >= node->distance) {
      munlock(node);
      kthr();
   }

   node->distance = distance;
   node->parent = from;
   munlock(node);

   for (edge_t e = node->edges; e != NULL; e = e->next) {
      switch(nthr()) {
         case -1:             //replication denied (sequential)
            dijkstra();
            break;
         case 0:              //replication allowed, current worker
            break;
         case 1:              //replication allowed, new worker
            dijkstra(e->head, node->distance + e->length, node);
            kthr();  //this worker dies upon completion
      }
   }
}
```

(b) Pre-processed source

```
   nthr r3,div$ok        #jump if division is allowed
      sequential code
   br div$end            #continue normal execution
   div$ok:               #on allowed division
   bne r3,div$1          #new worker goes to div$1
      worker0 code
   br div$end            #continue execution
   div$1:                #new worker code
      → New stack
      worker1 code
      → Restore stack
   kthr                  #new worker dies upon completion
   div$end:              #execution continuation
```

(c) Assembly (`switch` statement)

Figure 2. *Component version of Dijkstra*

component program + hardware support pair not only outperforms standard SMT (and superscalar) execution, but it is fairly stable since hardware contexts are almost always used. More precisely, whenever a component dies, either because it has reached an end node or because it has met a component with a shortest path, another component can explore more paths concurrently. We could implement and observe similar synergistic behavior between program and architecture in most of our component programs.

3. Implementation

In this section, we present the architecture and compiler support for component-based programming.

3.1. Architecture Support for Components

As mentioned in the introduction, our current target architecture is SMT. SMT is a natural hardware platform for

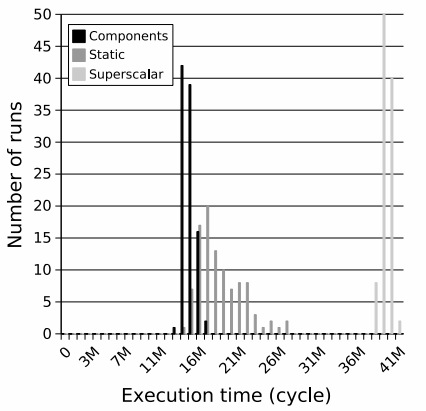

Figure 3. *Distribution of execution time (Dijkstra).*

evaluating component programming because a lightweight thread is a simple means for implementing a component. Other architectures, like CMPs, would also provide an adequate support, and we will extend our framework to CMPs in the future. A SMT must be augmented with three features to support component programming: (1) thread division, (2) thread activation/deactivation, (3) fast thread synchronization. The first item is component-specific, while the last two features have already been proposed for SMTs. Because the division feature enables the architecture to take parallelization decisions, we say the SMT is a self-organized multithreaded processor and called *SOMT*.

Our baseline SMT implementation is similar to the one proposed by Tullsen, et al., [38], see Figure 4. There are 8 hardware contexts, 32 registers per context, and 16 instructions can be fetched per cycle, using a policy similar to Icount 4.4 policy [37], 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. A thread may, by means of a `New THRead` instruction (`nthr`), divide itself into two new threads. One of the key features of the architecture is that it is free to ignore this instruction and not perform a thread division if available hardware resources do not 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 a worker/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 worker instructions are fetched. The thread registers are only copied at the commit stage because `nthr` could be speculative; it would also be

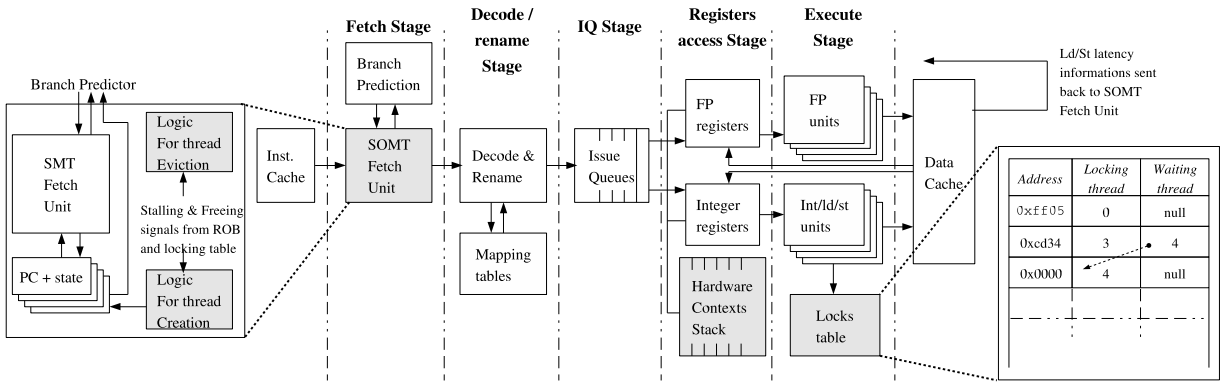

Figure 4. *Self-Organized Multi-Threading.*

possible (faster but more costly) to speculatively copy register map tables. The parent thread is stall during one cycle for the copy, and the child thread is stall a variable number of cyles until all register copies are done. 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 worker 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/workers in most cases, so this optimization did not seem worth the added hardware complexity.

A set of related workers, i.e., childs or parents or siblings, forms a group which always has a single ancestor; and each worker within such a group ends with a `Kill THRead` instruction (`kthr`), except for the ancestor. Upon decoding `kthr`, the corresponding thread transits from the `active` to `stall` state, and stops fetching instructions. When `kthr` reaches the commit stage, the hardware context is deallocated (`free` state).

Division strategy. As mentioned before, the architecture decides whether to act upon an `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 ($N = 128$ in our experiments) is smaller than half the number of hardware contexts.

Thread activation/deactivation. The proposed model also relies on the ability to swap in and out threads in order to have more threads (workers) 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 4. Tune et al. [40] already proposed to implement *virtual* hardware contexts using a context stack, called an *inactive register buffer*. Using micro-code instructions to save/restore registers, they estimate their swapping latency to no more than 15 cycles on average. Because we use no register mask and no fine-tuned swapping implementation, and because we use 16 virtual contexts instead of 4 (and 8 physical contexts versus 2), we estimated swapping at 200 cycles for 62 registers. We experimentally found that a LIFO stack of 16 entries was sufficient for an architecture with 8 hardware contexts. For 62 registers (31 FP, 31 Integer) plus a 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.

Scheduling and swapping strategy. The scheduling policy of SOMT is ICount.4.4 [37], 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.

Fast thread synchronization techniques. As proposed in [39], mutual exclusion for accessing shared variables is implemented using a fast locks table, see Figure 4. The lock is set by a `Memory LOCK` instruction (`mlock`) 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 `mlock` instruction wants to access a locked address, the subsequent instructions are squashed, the thread transits to the `stall` state and the thread id is stored in the *Locking table*, see Figure 4. 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, with a `Memory UNLOCK` instruction (`munlock`), the oldest waiting thread becomes the new owner.

3.2. Compiler Support for Component Programming

Rather than forcing the adoption of a new component language, we have implemented component support in the

form of C/C++ extensions that a user may or may not take advantage of, enabling progressive adoption, much like multimedia SIMD extensions. The role of the compiler support is to implement the C/C++ syntax extensions required for component programming and illustrated by the example of Figure 2(a), and to generate a code that can be manipulated by the architecture, especially for replicating components/workers. In the remainder of this section, we use our syntax term *worker* rather than *component*.

Our toolchain takes the form of a source-level pre-processor, an assembly-level post-processor, both combined with `GCC`. The source-to-source pre-processor transforms our C/C++ extensions for division into standard C/C++ code, as shown in Figure 2(b) for the example of Figure 2(a). This code is fed to `GCC`, and the corresponding assembly code is post-processed to substitute the source constructs introduced by the pre-processor with binary libraries for component programming and add several new assembly instructions.

One of the key transformations is enabling a worker to (efficiently) replicate or not, depending on the architecture decision at run-time. For that reason, any call to a C/C++ function identified as a `worker`, see Figure 2(a), is replaced with the `switch` statement shown in Figure 2(b) by the pre-processor. Three different versions of the worker are generated: a sequential version, left and right versions which split the worker task and its data structures in some cases.[3] The assembly post-processor will replace the assembly version of this switch statement with a more efficient assembly code, and introduce a `nthr` (`New THRead`) assembly instruction for the architecture, see Figure 2(c). If the architecture allows replication, the left/right versions are executed, otherwise, the sequential version is executed.

Now, replicating a worker does not amount to a standard function call, especially with respect to stack management. Because the left/right versions will execute concurrently, they cannot share the same stack, so a new stack is allocated from a pre-allocated pool in one of the versions (the right version; the left version is actually the initial worker with a reduced task).[4] This stack management code adds a slight overhead to replication, in addition to the hardware overhead of replication detailed in Section 3.1; the measured average programming overhead is 15 cycles per division.

Finally, replicating also requires to set synchronization locks on variables passed to co-workers. Workers abide by a simple rule enforced by the architecture: only one thread can execute on one or a user-specified set of data structure nodes. If another thread wants to use one or several data structures nodes used by other workers, it is simply stalled until the lock is released.

The architecture provides support for fast synchronization/locks as explained in Section 3.1. By default, locks are set by the toolchain on every variable passed by address; they are only set on the sections where variables are modified, and they do not include spawning sections. Intuitively, the principle is to set locks systematically when global data is manipulated, but to release these locks before any worker "movement" on the data structures. However, the toolchain does not include sophisticated inter-procedural or alias analysis and the user must still adjust locks manually when necessary. More generally, in its current form, the approach does not relieve the user from identifying deadlocks, and lock placement can be modified by the user. For instance, in Figure 2(a), the lock on variable `from` is not necessary and has been removed; variable `node` is unlocked before the `for` loop, where the worker "moves" to child nodes (spawning sections). If a worker reaches a graph node currently scanned by another worker, it will stall until that worker has either updated the node distance and moved to neighbor nodes or died.

We are currently working on expanding the framework syntax with special constructs for identifying data structures and data structure manipulations which will enable the toolchain to implement data-centric synchronization, and to handle locks in more cases, putting less emphasis on the need for sophisticated inter-procedural and alias analysis. This data-centric synchronization will itself be based on protected objects. Protected objects are standard objects where only a single method can be executed at any time. This approach is based on Ada protected objects [42], and the core concept itself comes from Monitors [16]. When coupled with tight data encapsulation within objects along the principles of components, it facilitates the identification of deadlocks and the management of concurrency. By default, the software toolchain automatically generates locks, but again, the user is free to alter the model and unprotect some methods to ensure correct execution. The data structures and data structure manipulations will later be based on protected objects and dedicated methods within these objects.

It is important to understand that, while synchronization is usually a strong performance limitation of parallel applications, component programs usually have no central data structure that all workers need to access *simultaneously*. Component programs reflect the notion that data structures are laid out on a virtual space with multiple workers accessing their different nodes concurrently, so that, even though synchronizations occur all the time, they only occur among a few workers on a given data structure node. As a result, the impact of synchronizations on performance is fairly low, and it does not increase significantly with the number of workers, i.e., when the number of available hardware resources/threads increases. Finally, note that in the *Dijkstra* example, each worker uses only one node of one data structure, but in other programs such as *LZW*, a worker uses multiple nodes from two different data structures, i.e., the approach does not limit how many data structures a worker can handle, nor how many nodes of each data structure.

The final support for replication is the process of killing a worker. Upon death, a worker must merge/combine its information with co-workers, and only one of them will survive. The nature of worker groups is program-dependent: all workers may belong to the same group (as in *Dijkstra*) or several worker groups may independently co-exist. In some cases, like a reduction for instance, this merger involves special processing, such as adding the worker result with that of other co-workers. Progressively combining local results from co-workers rather than updating a central

[3] For instance, replicating a worker which uses an array may involve changing loop bounds.

[4] Note that for small workers where only registers are used, there is no need to allocate a new stack upon each division.

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

Table 1. *Baseline configuration of SOMT, SMT and superscalar processors.*

variable or data structure significantly improves program parallel properties. Note that the `REDUCTION` OpenMP directive provides a similar specific support for reductions through the addition of local variables.

4. Methodology

Simulator. Our SMT simulator is built on top of SimpleScalar version 3.0. The functional simulator part of SimpleScalar was replicated to accomodate multiple threads; modifications also involved a significant rewrite of the fetch stage (see below), and lesser modifications of the dispatch stage (replication of mapping tables). We ran experiments on an SOMT processor, a SMT processor, and an aggressive superscalar processor configured according to the parameters in Table 1. While Capsule can potentially accomodate several processes and leverage dynamic component division to take advantage of periods of time with low system workload, multi-process workloads have not been implemented nor evaluated yet, and are left for future work.

Instruction fetch. For each thread, one cache line (i.e., up to 8 instructions) is fetched from the instruction cache to the instruction buffer. For the SMT/SOMT versions, 4 cache lines, corresponding to 4 different threads, are fetched every cycle, i.e., up to 32 instructions. Two branch predictions can be performed every cycle, and only one prediction per cache line per cycle. We use a double 16-instructions buffer to store the fetched instructions, but only the first 16 instructions can be used in a single cycle. The instructions are stored in the second buffer solely to reduce the number of cache accesses and to increase the overall fetch bandwidth. Every cycle, only 4 threads can fetch instructions from the buffer, and the instructions of the different threads are interleaved in the buffer, but the repartition of instructions among threads is dynamic. If one thread can fetch less than 4 instructions on a cycle, the instructions of other threads are stored in the buffer. Therefore, while, on average, 4 threads get 4 instructions each per cycle, one thread can get up to 8 instructions and issue them all in a single cycle, see Table 1.

Benchmarks. Our benchmark suite contains 4 SPEC CINT2000 programs, indicated in Table 2, and 4 core

SPEC CINT2000	# Lines	# Modified or added functions	Modified or added lines	% total execution time
`181.mcf`	2412	2	174	45%
`175.vpr`	17729	10	624	93%
`256.bzip2`	4649	3	317	20%
`186.crafty`	45000	8	201	100%

Table 2. *SPEC CINT2000 modifications for componentization.*

algorithms, 2 of which are used in other SPEC CINT2000 benchmarks (`179.art, 164.gzip`). We coded the algorithms into independent components: the *LZW* compression algorithm (`164.gzip`), the *Dijkstra* routing algorithm (`175.vpr`), the *Perceptron* neural network (`179.art`), the *QuickSort* sorting algorithm (`181.mcf`, `256.bzip2`). The execution times of the different componentized algorithms vary from several hundred thousand cycles to several hundred million cycles depending on data set size.

As mentioned in Section 2, a component implementation of a given program specification can be significantly different from its C/C++ implementation. For that purpose, in order to extract the full program specification, we had to reverse-engineer the target program specification and then re-engineer program sections using components. Because it is a very time-consuming and tedious process, we have only applied it to 4 SPEC CINT2000 programs. For each benchmark, we have identified a subset of the control-flow graph where the program "stays" (consecutively spends) a significant share of its execution (so that data structures may be modified within these subgraphs without incurring too frequent and excessive performance penalties when switching back to the rest of the program). Then we have identified and reprogrammed in a component way the corresponding sets of functions. Table 2 indicates for each benchmark the number of modified source lines and functions, and the total execution time spent in the corresponding subgraph.

Static parallelization. In order to highlight the synergy between component programming and component hardware support, we have derived a statically parallelized version of each core algorithm to be run on a standard SMT, in addition to the baseline imperative version to be run on a superscalar processor, and the component version to be run on SOMT. All benchmarks were compiled on an Alpha 21264 using `cc -O3`.

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*,[5] *Dijkstra*,[6] *MCF*). In order to have a homogeneous statically parallelized version of all programs, we have derived a static parallel version for each program from our component version using profile-based techniques. The general principle is akin to iterative parallelization: we run the component version, monitor how data structures are implicitly being divided by workers, and whenever the number of workers reaches the maximum number of hardware contexts, we record how the data is distributed among work-

[5] Derivatives of *LZW* have been parallelized but not *LZW* itself.

[6] The *Dijkstra* parallel versions are intended for very coarse parallelism, and would not perform efficiently on small data sets.

0-7695-2732-9/06 $20.00

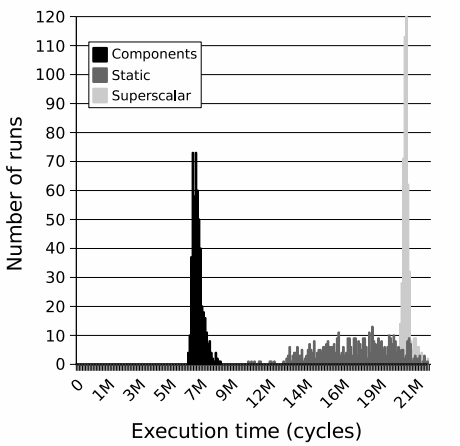

Figure 5. *Distribution of execution time (QuickSort).*

ers, and use this distribution as a static task parallelization. Therefore, the comparison of our component version against our static version is rather pessimistic 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.

5. Performance Evaluation

We first use the core algorithms to illustrate a number of properties of our hardware/software component approach, and then apply it to the re-engineered SPEC CINT2000 programs.

Core algorithms. As mentioned in Section 4, in most of the 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 a component version dynamically parallelized on a self-organized SMT.

Irregular data structures and parallelism. One of the first benefits of component hardware support is dynamic load balancing. Our division strategy greedily grants component spawning requests as long as hardware contexts are available. If an algorithm needs to partition complex and irregular data structures, the workload of each thread may significantly vary. With static parallelization, a thread with a small workload which terminates early will stay idle. With the component approach, all components will constantly probe the architecture for division through the `nthr` instruction. As soon as a worker (thread) has terminated, its hardware context will be granted to one of the remaining components, so that hardware resource utilization is maximized.

This effect shows in Figures 3 and 5 for both *Dijkstra* and *QuickSort*; respectively 100 graphs of 1000 nodes were generated for *Dijkstra* and 500 lists of various distributions for *QuickSort*; the x-axis shows the execution time, and the y-axis is the number of data sets with the same execution time. These figures show both the performance difference as well as the performance variability of each

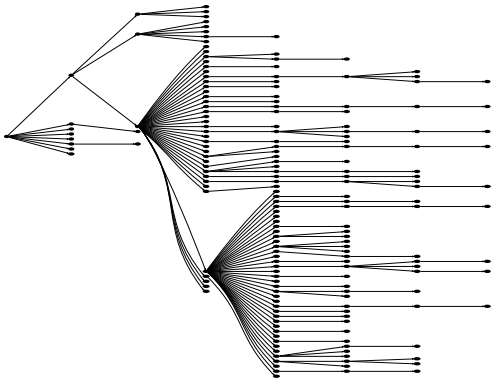

Figure 6. *Irregular divisions in QuickSort.*

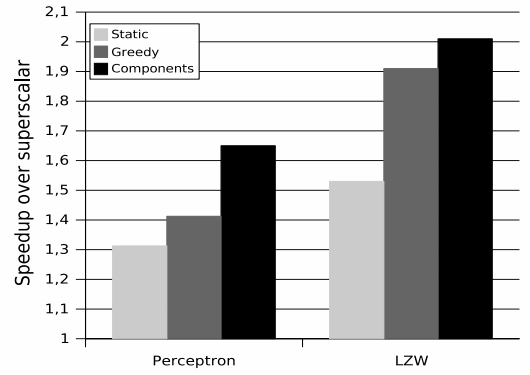

Figure 7. *Division throttling of small parallel sections.*

approach. *Dijkstra* partitions the graph into as many subgraphs of varying size as the number of child nodes after each node selection, and *QuickSort* partitions the list to sort into two sub-lists of varying sizes depending on the pivot selection. The component version has a speedup of 1.23 for *Dijkstra* and 2.51 for *QuickSort* over the statically parallelized version, and respectively 2.51 and 2.93 over the superscalar version. Its performance is also significantly more stable than the statically parallelized version due to dynamic load balancing, which has interesting implications for embedded real-time systems. Figure 6 illustrates for *QuickSort* how irregular divisions can be depending on the list lengths after each pivoting operations.

Small parallel sections. Greedy component division improves dynamic load balancing by maximizing hardware resource usage. However, as mentioned in Section 3.1, if the lifespan of a worker (thread) is short, the overhead will eradicate the benefits of exploiting parallelism. For that reason, 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. *LZW* and *Perceptron*, are two algorithms which benefit from this division throttling feature.

The *Perceptron* component version constantly attempts to split its initial group of 10000 neurons into two child components with half the number of neurons. The *LZW* component version recursively splits the initial sequence of $N = 4096$ characters it must match into two sequences of $N/2$ characters in order to parallelize the search.

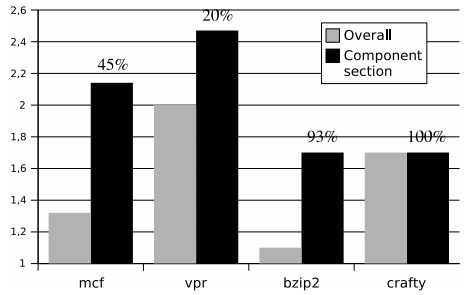

Figure 8. *Speedup (overall and component sections; % above bars correspond to fraction of total execution spent in componentized sections).*

Benchmark	# divisions requested	# divisions allowed	% divisions allowed	# insts / division allowed
mcf	99598	40532	40%	3.7K
vpr	67560	2702	4%	4.5M
bzip2	38656	2319	6%	30M

Table 3. *Percentage and rate of successful divisions.*

Because both components perform little processing on their data and have frequent opportunities to split, Figure 7 shows that they benefit from dynamic division throttling.

Re-engineered SPEC CINT2000. Most of the performance assets of our component approach have been illustrated with core algorithms in the previous paragraphs. The purpose of this section is twofold: (1) to demonstrate the approach on larger applications, and (2) to highlight that even complex applications like SPEC CINT2000 embed significant parallelism, at least at the *specification* level. As explained in Section 4, we had to reverse-engineer the whole programs in order to extract their specification, even though, in the end, we sometimes re-engineer/componentize only a small part. We have applied our component approach to SPEC CINT2000 programs because there are few examples of parallel versions of these benchmarks; however, directly writing components programs is both faster and yields more efficient programs. We report the performance of the component version on an 8-context SOMT versus a superscalar processor with the same resources; unless otherwise mentioned, the hardware configuration of the SOMT and superscalar processors is the one indicated in Table 1.

Figure 8 shows the overall speedup obtained for the default architecture, the speedup of the componentized section, and recalls the percentage of execution time spent in component sections, already indicated in Table 2. Table 3 indicates the number of divisions requested, the number of divisions allowed (recall divisions are not allowed if all hardware contexts are used) and the rate of divisions allowed (exactly the number of instructions per division). These statistics are an indication of the amount of available parallelism and how often components can saturate hardware contexts. We briefly describe the component part of each program below, and provide a few additional details on the specific behavior of each benchmark.

In `181.mcf`, the component replaces a sequential tree traversal (for route planning) with a parallel tree search. Note that the rate of divisions of `181.mcf` is significantly higher than for the two other benchmarks (though a division occurs only every 3679 instructions on average) because we chose to test division at every tree node, and because the code only performs an elementary task at each node. While a coarser division level could have been chosen, this example also shows that performance benefits can be achieved even when divisions are implemented at a very fine granularity.

In `175.vpr`, the component implements FPGA routing and placement by simultaneously exploring many circuit graph paths (up to 8000). The parallel version of the algorithm converges in 9 iterations instead of 8 for the original version, which decreases the overall speedup. The parallel version is memory bandwidth-limited, so doubling cache size and cache ports improves the speedup of a single iteration from 2.47 to 3.5, and the overall speedup to 3.0.

In `256.bzip2`, a block-sorting compression algorithm, the component targets the string sorting process.

In the `186.crafty` chess program, the component version is derived from an existing parallel implementation[7] which uses the `pthread` library [26] for splitting the search tree. The purpose of this example is to illustrate: (1) that component-based programming is compatible with existing parallel implementations, and (2) that static management of parallel hardware contexts is not as efficient as the dynamic management as in SOMT. The program maintains a pool of threads in active wait and, in some sense, manages thread contexts by software, and mostly inhibits dynamic component division. As a result `186.crafty` is not mentioned in Figure 3. Moreover, because there is an overhead for maintaining such a pool of pthreads, increasing the number of threads can actually degrade performance. The overall speedup of the same application on a 4-context SOMT is 2.3 instead of 1.7 for an 8-context SOMT.

Potential impact of CMPs on dynamic spawning. In this paragraph, we explore how porting our approach to CMP could affect its behavior and performance. We have not yet simulated a CMP version of our hardware support, we only extrapolate based on our SMT implementation. The two most critical parameters of our approach are the division/spawning latency and the probing latency.

On a shared-memory CMP, the probing latency is not expected to change significantly, but the division latency can vary more greatly due to the cost of starting a thread on a remote processor. We have simulated division latencies up to 200 cycles, and observed an average performance variation of less than 1% with respect to the results presented in this section. The main reason for this reduced impact of the division latency is that, in fact, the actual division rate remains limited. For instance, in `181.mcf`, which has the highest ratio of successful divisions among the SPEC CINT2000 programs we evaluated, there is only one division every 3679 insts on average. The approach does not necessarily require a high number of divisions to perform well, but rather the ability to adapt to irregular workloads, as illustrated in Figure 6. Frequent probing also enables rapid adaptation to changing conditions. Still, for significantly higher division latencies, we can naturally expect codes with high division rates such as `181.mcf` to

[7] Version 19.19 versus 14.3 in SPEC CINT 2000; the main algorithm is the same in the two versions, but several heuristics were modified, such as in scoring mechanisms.

benefit less from the division mechanism.

We are also investigating distributed-memory CMPs which are popular in embedded systems. In that context, we expect a more complex hardware support will be necessary, especially for probing, which will require fast communications between neighbor processors.

6. Related Work

Architectures for improving single-process performance. Many other research works propose to use single-chip multi-threaded architectures, such as CMPs or SMTs, to speed up single processes. For instance, Tullsen et al. [22] evaluates static parallelization on an 8-context SMT and achieves an average 2.68 speedup on 5 parallel SPLASH-2 benchmarks and 2 SPEC-95 benchmarks. Another approach for taking advantage of multiple threads to speedup single processes are *helper threads* [29], which spawn a reduced version of the main process, capable of running ahead of it, in order to warm up the cache or to provide a feedback on branch behavior.

Snavely et al. [31] proposes to co-schedule jobs that perform well together on a SMT. While this work focuses on multi-process workloads rather than improving the performance of a single process through parallelization, it demonstrates the potential synergy between software layers (the OS in this case) and micro-architecture support.

The micro-architecture support of SOMT shares several features with *Balanced Multithreading* proposed by Tune et al [40]. They implement a context stack that can hold idle threads so that the number of virtual threads can be significantly larger than the number of hardware contexts. The processor can quickly swap threads in and out, and the swapping strategy is based on their memory behavior. However, this work focuses on multi-process workloads again, so it cannot improve single-process performance other than by relying on a parallelizing compiler.

Wish Branches [19] bear some similarity with our approach because it delegates at run-time the decision to choose between a normal and a predicated branch; the decision depends on the quality of prediction.

The Network-Driven Processor proposed by Chen et al. [10] also shares several features with CAPSULE, and its initial version called *Agent Programming+SOMT* [21]. They similarly propose to delegate thread creation decisions at run-time to the hardware, and show an example application on a CMP; there is also a significant development on streaming applications and support. However, with respect to thread management, NDP is more like the hardware implementation of the Cilk run-time system: all thread spawning commands are effected, stored in a table and spawning stops when the table is full. There is no constant probing of hardware resources and conditional spawning. Furthermore, components seem to spawn childs of a given granularity, e.g., loop iterations, rather than gracefully divide in half until resources are full or the granularity is too small.

GPA/TRIPS [30], RAW [35] or WaveScalar [33] propose several innovative approaches to take advantage of on-chip space. GPA is attractive because it speeds up computations by mapping them on hardware instead of parallelizing them, and thus imposes a small burden on the compiler; however, its scalability may be limited. RAW favors, but is not restricted to, dataflow-like computations, and the StreamIt [36] compiler and language is focused on streaming applications for RAW. WaveScalar has a larger application scope but it still relies on compile-time automatic parallelization.

Speculative parallelization. Because of the limitations of compiler-based parallelization, thread-level speculation (TLS) has received significant attention in the past few years. Prabhu et al. [27] explored the potential of TLS on several SPEC CPU2000 benchmarks on the Hydra multiprocessor. The Mitosis compiler and architecture [28] is one of the most recent and thorough attempt. It proposes complex but sophisticated hardware support combined with compiler-based precomputation of threads live-ins. However, it is difficult to know whether this scheme can scale up to large amounts of parallelism. Steffan et al. [32] proposed a scalable hardware support for TLS for shared-memory multiprocessing systems. Program Demultiplexing [5] recently introduced an approach where parallelism is exploited at the procedure level, and procedures are speculatively spawned earlier than in a sequential execution. A hardware support helps buffer the program state modifications until the speculation is resolved.

Parallel programming and languages. There is such a large body of research works on expressing parallel semantic, that it is just impossible to convey them all here. Classical UNIX semantics (`fork()`, posix threads) and MPI are widely used but they are error-prone and hard to debug. Approaches like Hood [7] or Continuations [15] help hiding low-level primitives but still let the programmer handle concurrency accesses. To circumvent this difficulty, parallel programming paradigms have proposed to let the user express specifications using implicit concurrency [11, 4, 20]. However, many of these paradigms are domain-specific (scientific computing for OpenMP, spatial organizations for MGS, etc...). Intentional Programming, Generative Programming and Language Oriented Programming [41] aim at further facilitating the creation, usage and interoperability of parallel paradigms; they use domain-specific languages [6] to easily express programs, and then automatically generate standard code. X10 [9] and Fortress [1] have proposed implicit transactions and weak atomicity which help the programmer express the concurrent constraints of her code through dedicated language modifications. Speculative Synchronization [24] circumvents expensive lock checking through an optimistic policy with fallback. Both approaches require to implement a rollback mechanism. STAPL [3] is a different approach, more a programming framework than a language, for both easily designing parallel applications and simultaneously confronting the associated software engineering issues; the principle is to build a parallel version of the C++ STL library combined with the appropriate run-time support. TSM (Time-Shifted Modules) [43] proposes a similar library-based approach which further relieves the user of programming tasks at the expense of creating programs with slightly less and coarser parallelism. The principle is to leverage the fact that many programs are built as a set of mostly independent tasks. By building a program as a set of modules providing data encapsulation, it is possible to let the programmer reason al-

most sequentially on its program, while taking advantage of the parallelism between modules when several hardware contexts are available.

Our approach is more related to component programming environments such as Cilk and Charm++ mentioned in the introduction. Beyond the spawning support for parallelism, the separation property of components and their explicit communications provide an implicit hardware model. This model corresponds to a virtual space where components reside and are linked with each other. This model is significantly different from the Von Neumann model (processor/memory view) implicitly used by all programmers. It is more amenable to parallelism (two components residing in the space at the same time and not communicating implicitly execute concurrently), and shares properties with spatial computing models such as Blob Computing [12]. This implicit model can somewhat help the programmer for the task of extracting parallelism, though just rewriting a program into components is usually not enough to parallelize it.

7. Conclusions and Future Work

We have introduced a combination of component-based programming and component hardware support as a pragmatic approach for harnessing the parallelism in programs with complex data structures and control flow. By delegating component spawning and mapping decisions to the architecture, we have both simplified the task of the programmer, and achieved a better exploitation of hardware resources.

Furthermore, we have observed that, because this runtime component allocation strategy follows a simple rule of maximizing resource occupation, it has the side-effect of making the execution time of programs with complex behavior significantly more predictable. This property has interesting implications for real-time embedded systems.

Instead of proposing a new language or complex syntax extensions to write parallel programs, we have essentially leveraged (and are thus compatible with) a current software engineering trend towards greater encapsulation through component programming, and improved upon past research works combining components and parallel programming. Our component syntax can be adapted to JavaBeans, .Net or other component frameworks. It is also interesting to note that, intuitively, this current software engineering trend blends quite well with the current architecture trend towards multi-cores: both consist in breaking down respectively a complex program and a complex architecture into several more simple pieces. We essentially proposed a method for efficiently marrying them together.

Future work will focus on applying our component-based approach to shared-memory and distributed-memory CMPs. We do not foresee significant evolutions in the programming approach to tackle shared-memory CMPs; on the other hand, the hardware support must be augmented with probing neighbor nodes on thread availability, and thread migration. For distributed-memory CMPs, the programming support will encapsulate not only code sections but also data structures into separate components. We believe this can be achieved without significantly extending our current syntax, except spawning will affect both code and data components.

Acknowledgments

We would like to thank Sami Yehia, from ARM, for his support and many helpful suggestions.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification., 2005.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, pages 1–6. ACM Press, 1990.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, page 10, July 2001.

[4] H. Bal. Orca: a portable user-level shared object system, July 1996. Technical Report IR-408, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam.

[5] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methos in sequential programs. In *Proceedings of the 31st annual international symposium on Computer architecture*, June 2006.

[6] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[7] R. Blumofe and D. Dionisios. Hood: A user-level threads library for multiprogrammed multiprocessors, 1999. Technical Report, University of Texas at Austin.

[8] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[9] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, V. Saraswat, V. Sarkar, and C. V. Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.

[10] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L.-S. Peh, and M. Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News, Special Issue: Proc. of the dasCMP'05 Workshop*, 33(4):54–63, 2005.

[11] 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.

[12] F. Gruau, Y. Lhuillier, P. Reitz, and O. Temam. Blob computing. In *Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 125–139. ACM Press, 2004.

[13] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, New York, NY, USA, 1995. ACM Press.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual*

International Symposium on Computer Architecture, page 102. IEEE Computer Society, June 2004.

[15] R. Hieb and R. K. Dybvig. Continuations and concurrency. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 128–136, New York, NY, USA, 1990. ACM Press.

[16] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[17] J. Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, Sept. 2002.

[18] L. V. Kale and S. Krishnan. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 91–108. ACM Press, Sept. 1993.

[19] H. Kim, O. Mutlu, Y. N. Patt, and J. Stark. Wish branches: Enabling adaptive and aggressive predicated execution. *IEEE Micro*, 26(1):48–58, 2006.

[20] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, New York, NY, USA, 1993. ACM Press.

[21] Y. Lhuillier, P. Palatin, and O. Temam. Ap+somt: Agent-programming combined with self-organized multithreading. In *Workshop on Complexity-Effective Design*, June 2004.

[22] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. 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.

[23] D. B. Loveman. High Performance Fortran. *IEEE Parallel Distrib. Technol.*, 1(1):25–42, 1993.

[24] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 18–29, New York, NY, USA, 2002. ACM Press.

[25] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117. ACM Press, 1995.

[26] F. Mueller. A library implementation of POSIX threads under Unix. In *Proceedings of theWinter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, 1993.

[27] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 142–152, New York, NY, USA, 2005. ACM Press.

[28] 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 *PLDI '05: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM Press, 2005.

[29] 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.

[30] 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.

[31] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 66–76. ACM Press, 2002.

[32] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, pages 1–24, 2000.

[33] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[34] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[35] M. Taylor, W. Lee, J. Miller, D. Wentzlaff, B. Greenwald, V. Strumpen, N. Shnidman, I. Bratt, H. Hoffmann, J. Kim, A. S. J. Psota, Jonathan, M. Frank, S. Amarasinghe, and A. 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.

[36] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[37] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. 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.

[38] D. M. Tullsen, S. J. Eggers, and H. 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.

[39] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. 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.

[40] E. Tune, R. Kumar, D. Tullsen, and B. Calder. Balanced Multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, Dec. 2004.

[41] M. Ward. Language oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

[42] A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Integrating object-oriented programming and protected objects in ada 95. *Ada Lett.*, XXII(2):11–44, 2002.

[43] C. Zilles and G. Sohi. Time-Shifted Modules: Exploiting Code Modularity for Fine Grain Parallelism. Technical Report TR1430, University of Wisconsin-Madison, Computer Sciences Dept., Oct 2001.