The compiler for the XMTC parallel language: Lessons for compiler developers and in-depth description

Alexandros Tzannes, George C. Caragea, Uzi Vishkin, Rajeev Barua

University of Maryland, College Park
Department of Computer Science
&
University of Maryland Institute for Advanced Computer Studies

CS-TR-4972
UMIACS-TR-2011-01

Revised: December 29, 2012

Abstract

In this technical report, we present information on the XMTC compiler and language. We start by presenting the XMTC Memory Model and the issues we encountered when using GCC, the popular GNU compiler for C and other sequential languages, as the basis for a compiler for XMTC, a parallel language. These topics, along with some information on XMT specific optimizations were presented in [10]. Then, we proceed to give some more details on how outer spawn statements (i.e., parallel loops) are compiled to take advantage of XMT's unique hardware primitives for scheduling flat parallelism and how we incremented this basic compiler to support nested parallelism.

1 Overview

The XMTC compiler translates XMTC code to an optimized XMT executable. Roughly speaking, the XMTC compiler consists of three consecutive passes: the *pre-pass* performs source-to-source (XMTC-to-XMTC) transformations and is based on CIL [14]; the *core-pass* performs the bulk of the compilation and is based on GCC v4.0; and the *post-pass*, built using SableCC [6], takes the assembly produced by the core-pass, verifies that it complies with XMT semantics (and amends it if necessary) and performs linking. The *xmtcc* bash script links the passes together for convenience and accepts multiple XMTC files, data files, and libraries as arguments.

One unusual aspect of the compilation of XMTC code is that assembling (converting assembly to binary) happens after linking. This choice was made to allow targeting both the XMT FPGA hardware [19, 20, 18], which accepts binary executables, and the XMT cycle-accurate simulator [10, 9], which accepts a big monolithic assembly file. This choice was made to make it easier to use the simulator as a debugging tool and to allow adding new instructions in the simulator for experimentation, without having to define a binary representation for them (opcodes, instruction formats, etc).

Figure 1 gives an overview of the compilation process of an XMTC file. We kept the `.c` file extension to have text editors interpret and highlight the code as C code, since XMTC is a modest extension of C. The pre-pass takes one or more XMTC source files and produces an intermediate `.cil.c` file for each source file. Then, the core-pass produces an assembly (`.s`) file for each of these intermediate files. Finally, the post-pass first performs some transformations on the assembly producing a `.p` file for each `.s` file, then links all the

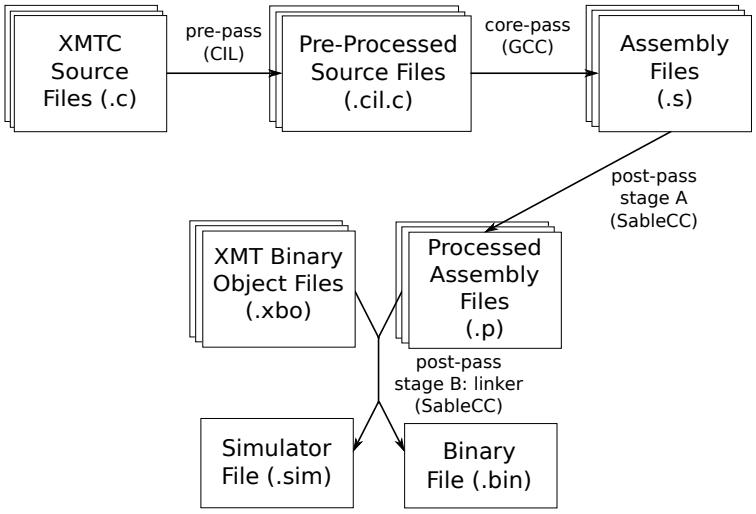

Figure 1: Compiler Passes.

files, possibly including library code and binary files (`.xbo`), and produces two files: a `.sim` file and a `.bin` file. The `.sim` file contains the instructions of the program in assembly and is used only when running the program on the simulator. The `.bin` file has different contents depending on whether the target platform of the compilation was the FPGA or the simulator. In the former case, the `.bin` file contains contains both the code and the initial data in binary format; in the latter case, it only contains the initial data in binary format, and the instructions are not included (they are replaced by zeros to maintain correct addressing of data).

The different passes of the compiler host different functionalities of the compilation process. Here, we briefly list the functionality that each of the three compiler passes implement and we expand on them later in this technical report.

The pre-pass (CIL) is home to source-to-source transformations, including: *(1) function cloning*[1] (Section 6.1) to keep stack management in sequential code optimal and to keep track of which spawn-statements are nested (Section 6); *(2) outlining of outer spawn statements* to prevent illegal data-flow across spawn-statement boundaries (Section 3.1); *(3)* spawn-block and spawn-statement *cost prediction* and *automatic coarsening* based on that cost-prediction, including picking a granularity parameter for spawn-statements and static or dynamic serialization (see [16] Section 3.5); *(4) flattening* of perfectly nested spawns to reduce the nesting depth and avoid scheduling overheads; *(5) outlining and conversion of nested spawn-blocks* to create closures for tasks that the scheduler will be called to execute (Section 6.6).

CIL had to be modified to allow using it for XMTC. First, we extended the lexer and parser to include the XMTC extensions, but also the concrete syntax tree produced by the parser to include a new type of statement nodes, spawn-statements. Another much trickier modification that was required had to do with CIL's design to hoist all variable declarations within a function to the top of that function. While correct for a sequential C program, the XMTC spawn-statement implies that the scope of a variable declared within the spawn-block are private to it. In particular, multiple instances of the variable may exist simultaneously, as many as the tasks created by the spawn-statement. For that reason, hoisting of variable declarations is illegal in XMTC, and the internal data-structures of CIL had to be modified to support local variable declarations for each block of code. Finally, the procedure that builds the concrete parse tree in CIL had to be modified to correctly account for local variable declarations.

The core-pass (GCC) converts the intermediate source code to assembly. In addition to all the conventional GCC optimization passes, the core-pass implements: *(1) live-register broadcasting* for transitioning

[1]Function Cloning simply creates a parallel clone of each function in the code and updates each call-site to invoke the appropriate clone.

from sequential to parallel mode (Section 3.2); *(2) cactus-stack allocation* to support a parallel stack for the parallel portions of the code (Section 6.2); *(3) global register* loading and reading, implemented in the XMT back-end in GCC and not described here; as well as *(4) linear and loop prefetching* implemented by George C. Caragea [3, 5, 4].

GCC's parser had to be modified to parse XMTC, but its internal data structures were not. Instead, the new constructs were expressed using GCC's existing data structures. This choice was made to allow the use of all of GCC's optimizations without having to update them to account for the new types of statements and in particular explicit parallelism. For example, a spawn-statement is compiled as a `spawn` inlined assembly instruction (GCC does not need to know about its semantics) that is in turn followed by the spawn-block, followed by a `join` inlined assembly instruction. This, of course, opens the door for illegal data-flow and code-motion, which are prevented by compiler passes, such as *outlining* (see Section 3). Furthermore, GCC's MIPS machine-description in GCC's back-end was cloned and incremented to create XMT's machine-description and to describe the existence of global-register, as well as rules for managing them. Additionally, XMT specific passes were added in this back-end, such as live-register broadcasting (Section 3.2) and cactus-stack allocation (Section 6.2).

The post-pass starts by performing a battery of simple transformations on the assembly files produced by the core-pass, mainly straightforward assembly sanity checks, rewriting, and simplifications. Then, it applies some more involved transformations, namely *function insertion* (Section 6.3), *dead function elimination* (Section 6.4), *assembly block reordering* (Section 3.3), *burst prefetching* (Section 4.2), *global address and label calculation*, *linking of data-files*, and *assembling*.

The post-pass was built from scratch using the SableCC parser generator. The grammar describing the language for parsing the assembly produced by the core-pass is an extended version of the grammar written by Fuat Keceli for his cycle-accurate XMT simulator [10, 9]. The reader is expected to have a basic understanding of the XMT architecture which can be acquired by reading the relevant sections in one of [10, 4, 9, 16]. Detailed information on the XMT architecture can be found in Wen's dissertation [18].

2 The XMTC Memory Model

The memory consistency model for a parallel computing environment is a contract between the programmer and the platform, specifying how memory actions (reads and writes) in a program appear to execute to the programmer, and specifically which values reading a memory location may return [1, 12].

Consider the example in Figure 2. If memory store operations are non-blocking, meaning they do not wait for a confirmation that the operation completed, it is possible for Task B to read {x=0 and y=1}. At first this *relaxed consistency* is counter-intuitive, but because it allows for much better performance by allowing multiple pending write operations, it is usually favored over more intuitive but also more restrictive models [1, 12].

Initially: `x=0, y=0`

Task A:	**Task B:**
x := 1	Read y
y := 1	Read x

Possible results read by Task B: $(x, y) \in \{(0,0), (1,0), (1,1), (1,0)\}$

Figure 2: Two tasks with no order-enforcing operations or guarantees.

The XMT memory model is a relaxed model that allows the same results for Task B as in the previous example. It relaxes the order of memory operations and only preserves relative ordering with respect to

prefix-sum operations (`ps` and `psm`), and to the beginning and end of `spawn` statements. This makes prefix-sum operations important for synchronizing between tasks, as will be shown in Figure 3.

The XMT memory model gives the programmer two rules about the ordering of (read and write) memory operations. *First, it guarantees sequential execution within one task*, which means that a read to a memory location will return the last value written to that location by the current task, provided it was not overwritten by a different task. Intuitively, read or write operations from the same source (TCU[2]) to the same destination (memory address) will not be reordered, neither by the hardware, nor by the compiler. This rule allows the programmer to treat each task as they would treat sequential code, as long as the tasks do not modify shared data (i.e., there are no data-races). The next rule deals with disambiguating (synchronizing) access and modification of shared data.

Second, for each pair of tasks, the XMT memory model guarantees a partial ordering of memory operations relative to prefix-sum operations over the same base. This rule allows the programmer to reason about *"happens before"* [11] relations and to enforce synchronization between concurrent tasks.

This rule is a bit more involved, so we explain it through the example in Figure 3. This example shows how to implement the example of Figure 2 if we want the invariant *if y=1 then x=1* to hold at the end of Task B (*i.e.*, disallow $(x, y) = (0, 1)$). Both tasks synchronize (in a loose sense) over variable `y` using a `psm` operation; task A writes (increments) `y` atomically whereas task B reads it. At run-time, one of the two tasks executes its `psm` instruction first; the second rule of the XMT memory model guarantees that all memory operations issued before the `psm` of the first task to execute its `psm` will have completed before any memory operation after the `psm` of the second task is issued. In our example, assume that task A completes its prefix-sum first. That means that the operation `x=1` completed before task B reads X, which enforces the desired invariant *if y=1 then x=1* for task B.

Initially: `x=0, y=0`

Task A:

```
x = 1;
tmpA = 1;
psm(tmpA,y); //y++
```

Task B:

```
tmpB = 0;
psm(tmpB,y);// Read y
.. = x       // Read x
```

Possible results read by Task B: $(x, y) \in \{(0, 0), (1, 0), (1, 1)\}$

Figure 3: Enforcing partial order in the XMT memory model.

The implementation of these two rules by the hardware and the compiler is straightforward. For the first rule, the static hardware routing of messages from TCUs to memory guarantees that the order of operations issued from the same source to the same destination will be preserved. The compiler enforces the second rule (a) by issuing a *memory fence* operation before each prefix-sum operation to wait for all pending writes to complete, and (b) by not moving memory operations across prefix-sum instructions. The current implementation does not take into account the base of prefix-sum operations and may be overly conservative in some cases. Using static analysis to reduce the number of memory fences and to selectively allow motion of memory operations across prefix-sums could be the topic of future research. It is unlikely, however, that such an optimization would yield substantial benefits since prefix-sum operations are typically used to synchronize between tasks. In the future, if more types of atomic operations are added to XMTC, the memory model may have to be updated to enforce partial ordering with respect to them as well.

Note that in Figure 3 both `psm` operations are needed. If, for example, Task B used a simple read operation for `y` instead of a prefix-sum, prefetching could cause variable `x` to be read before `y` and the invariant *if y=1 then x=1* would not hold.

[2]Thread Control Unit: A light-weight parallel processing unit in XMT.

An implication of the XMT memory model is that register allocation for parallel code is performed as if the code were serial. The programmer, however, must still declare variables that may be modified by other tasks as `volatile`. These variables will not be register allocated, as in the case of multi-threaded C code. This is rarely needed in XMTC user code, but it is useful in low-level library and system code.

Finally, the memory-fence operation is available to expert programmers who want to explicitly enforce ordering of specific memory operations because sometimes a prefix-sum operation is not ideal. In the example above, task A would simply set `x` to 1, invoke the memory-fence, then set `y` to 1. This would have the same effect as the code of Figure 3 but with the added benefit that `y` would be written using a non-blocking store operation, which is more efficient than the blocking `psm` operation.

3 Compiling XMTC Parallel Code with a Serial Compiler

Although we have extended the core-pass (GCC) to parse the additional XMTC parallel constructs, it inherently remains a compiler for sequential C. Changing the internal GCC data structures to express parallelism would have required great effort, and all the optimization passes would have had to be updated to account for these new constructs. Such a task was beyond the scope of this work and fortunately proved unnecessary in practice. Instead, a spawn statement is parsed as if there was a `spawn` inlined assembly instruction at the beginning of the spawn block and a `join` at the end of it. Figure 4(a) shows the code as written by the programmer, whereas Figure 4(b) shows how the compiler interprets it. Therefore, GCC interprets a spawn statement as a sequential block of straight-line code. This opens the door for illegal dataflow because *(1)* it hides the fact that the spawn block might be executed multiple times (i.e., it hides its loop semantics), *(2)* it hides the concurrency of these multiple executions, and *(3)* it hides the transfer of control from the Master TCU to the parallel TCUs where the spawn-block is executed in the case of an outer spawn statement.

An invalid code transformation caused by illegal dataflow is code-motion across spawn-block boundaries. For example, the code of Figure 4(a) reads all the elements of array A in parallel and, if an element is non-zero, it sets `found` to `true`. After the parallel section, `counter` is incremented if a non-zero element was found. The compiler may choose to move the conditional increment statement `if(found) counter+=1` before the `join` instruction to issue the non-blocking store operation for `counter` earlier and overlap it with the `join` instruction (whose semantics it does not know). The counter could then be incremented multiple times instead of only once, which breaks the semantics of the original program.

3.1 Outlining

To prevent illegal dataflow, we implemented *outlining* (also known as *method extraction*) in the CIL pre-pass, an operation akin to the reverse of function inlining. Figure 4(c) shows the outlined version of the code in Figure 4(a). Outlining places each spawn statement in a new function and replaces it by a call to this new function. According to XMTC semantics, the spawn statement should have access to the variables in the scope of the enclosing serial section, so the outlining pass detects which of these variables are accessed in the parallel code and whether they might be written to. Then, it passes them as arguments to the outlined function by value or "by reference" accordingly. In Figure 4(c), because the variable `found` is updated in the spawn block, a pointer to it is passed to the outlined function, and the spawn block is updated to access it through the pointer.

Outlining prevents illegal dataflow without requiring all optimizations to be turned off. This solution works because GCC, like many compilers, does not perform inter-procedural optimizations. Compilers that do perform inter-procedural optimizations often provide a flag that has the effect of preventing inter-procedural code motion.

(a) Original Code

```
int A[N];
bool found=false;
spawn(0,N-1) {
    if (A[$]!=0)
        found = true;
}
if (found) counter+=1;
```

(b) What the compiler sees

```
int A[N];
bool found=false;
asm(spawn 0, N-1);
if (A[$]!=0)
    found = true;
asm(join);
if (found) counter+=1;
```

(c) After Outlining

```
int A[N];
bool found=false;
outl_sp_1 (A, &found);
if (found) counter+=1;
.....................

void outl_sp_1(int (*A),
               bool *found) {
    spawn(0,N-1) {
        if (A[$]!=0)
            (*found) = true;
    }
}
```

Figure 4: Simple example of outlining.

3.2 Register Broadcasting

We now present another example of illegal dataflow, but this time without code motion. It happens because GCC is unaware of the transfer of control from the serial processor (Master TCU) to the parallel TCUs that a spawn statement entails on XMT. GCC optimizations incorrectly assume that a value can be loaded to a register before the spawn statement (within the outlined function) and later accessed within the spawn-block. This is not the case because the value is loaded to a Master TCU register while the spawn-block code accesses the TCU registers. There are two ways to fix this problem: (a) move the load instruction back into the spawn-block, causing each task to load the value from memory, potentially creating a memory hot-spot, or (b) broadcast all live Master TCU registers (an XMT specific operation) to the parallel TCUs at the onset of a task. We chose the second approach because it conserves memory bandwidth.

(a) Wrong layout by GCC

```
outlined_spawn:
    spawn
BB1:
    ...
    bneq $r, $0, BB2
    join
    jr $31  # return
BB2:
    ...
    j BB1
```

(b) Corrected layout

```
outlined_spawn:
    spawn
BB1:
    ...
    bneq $r, $0, BB2
    j BBjoin // GCC tried to
             //save this jump
BB2:
    ...
    j BB1
BBjoin:
    join
    jr $31  # return
```

Figure 5: Example of assembly basic-block layout issue.

3.3 Assembly Code Layout Correction

Finally, XMT places a restriction on the layout of the assembly code of outer spawn blocks, because it needs to broadcast the code to the TCUs. The restriction is that all spawn-block code must be placed between the `spawn` and `join` assembly instructions. Interestingly, in its effort to optimize the assembly, GCC might decide to place a basic-block (a short sequence of assembly instructions) that logically belongs to a spawn-block after it. In the example of Figure 5(a), basic-block 2 (BB2) is placed after the return statement of the `outlined_spawn` function to save one jump instruction.

The assembly code produced by GCC has correct semantics, but it will lead to incorrect execution on XMT because BB2 will not be broadcast by the XMT hardware, and TCUs do not currently have access to instructions that were not broadcast. Future versions of XMT will allow TCUs to fetch instructions that are not in their instruction buffer by including instruction caches at the cluster or TCU level.

One way to avoid this code layout bug would be to disable the offending optimization passes, but that would prevent the optimizations from happening even when they are legal. Instead, the post-pass checks for layout violations and fixes them by relocating misplaced basic-blocks, as shown in Figure 5(b). Note that after this relocation, it would be potentially beneficial to perform a peephole optimization to reduce jump instructions. In Figure 5(b), the two jump instructions in BB1 can be replaced by a single jump instruction: `beq $r, $0, BBjoin`.

3.4 Why illegal dataflow is not an issue for thread libraries

There are several libraries that are used to introduce parallelism to serial languages (e.g., *Pthreads*). Code written using such library calls are compiled using a serial compiler, so one might wonder why illegal dataflow is not an issue in that scenario. In Pthreads, the programmer creates an additional thread using the `pthread_create` call, which takes as an argument a function to execute in the new thread. In other words, the programmer is forced to do the outlining manually. Moreover, thread libraries do not introduce new control structures in the base language, such as XMTC's spawn statement, so the compiler does not need to be updated. That said, serial compilers can still perform illegal optimizations on Pthreads code [2], but these are rare enough so that Pthreads can still be used in practice. The main disadvantages of using a thread library, however, is the lack of compiler optimizations specifically targeting parallel code and the added complexity for the programmer: creating parallelism through a library API is arguably harder and less intuitive than directly creating it using parallel constructs incorporated in the programming language. The converse argument, that libraries are preferable to new languages, has also been stated [17], primarily on the basis of backward compatibility with sequential code-bases and of the reluctance of programmers to learn new languages. We believe that parallel extensions to existing languages are preferable because they provide backwards compatibility, cleaner and easier parallel coding, and better performance. We concede, however, that extending a programming language is a much more substantial effort than writing a library, but we believe it is worth the effort.

4 Latency tolerating mechanisms.

The XMT memory hierarchy is designed to allow for scalability and performance. To avoid costly cache coherence mechanisms (in terms of chip resources as well as memory bandwidth overheads), the first level of cache is shared by all the TCUs, with an access latency in the order of 30 clock cycles for a 1024 TCU XMT configuration. Several mechanisms are included in the XMT architecture to overlap shared memory requests with computation or to avoid them: non-blocking stores, TCU-level prefetch buffers, and cluster-level read-only caches. Compiler managed scratch-pad memory per TCU or per cluster is also on XMT's roadmap.

Currently, the XMT compiler includes support for automatically replacing eligible writes with non-blocking stores and for inserting prefetching instructions to fetch data in the TCU prefetch buffers. Support for automatically taking advantage of the read-only caches is planned for future revisions of the compiler. In the meantime, programmers can explicitly load data into the read-only caches if needed.

The XMT compiler offers three prefetching options: linear prefetching, loop prefetching, and burst prefetching. The first two were developed by George C. Caragea as part of his dissertation [4], whereas burst prefetching was contributed by Alexandros Tzannes to help support function calls in parallel mode efficiently.

Linear prefetching issues a prefetch instruction for each memory load and the modified instruction scheduler pass in GCC tries to hoist the prefetch instruction up the control-flow graph (CFG) to hide as much latency as possible. The lack of local caches on XMT makes linear prefetching very profitable as it compensates for the lack of spatial locality normally provided by caches. Linear prefetching is not resource-aware, and in some cases, it can degrade performance by thrashing the prefech buffer with requests that overwrite one another. This happens, for example, when more prefetches than prefetch buffer locations are issued simultaneously.

The loop prefetching mechanism was designed to match the characteristics of a lightweight, highly parallel many-core architecture. It has been shown to out-perform state-of-the-art prefetching algorithms such as the one included in the GCC compiler, as well as hardware prefetching schemes [3]. The key observation is that taking into account the size of the prefetch buffer and reducing the prefetch distance accordingly benefits performance. The prefetching algorithm is potentially applicable to other many-core platforms with small prefetch buffers. In the next section, we describe loop prefetching in detail.

4.1 Resource-Aware Prefetching

In this section, we describe a new compiler prefetching algorithm – Resource-Aware Prefetching (RAP) [3] – which improves upon Mowry's loop prefetching algorithm [13] as well as the GCC implementation by taking into account how few *Miss Handling Architecture* (MHA) resources are available and using them more efficiently. Our algorithm robustly adapts to constrained resources and uses them to hide as much latency as possible. More concretely, we show that in situations where not enough prefetch slots are available to issue prefetch instructions for all references, it is more beneficial to decrease the prefetch distance and prefetch for as many references as possible. By contrast, the GCC implementation uses a fixed prefetch distance and may prefetch fewer references.

Consider the code in Figure 6 as our running example. Figure 6(a) shows the original program code. In the figure, assume that the matrices A, B and C contain double precision floating point elements (64 bits) and our hypothetical system has a cache line of 16 bytes; thus two doubles fit per cache line. Also, assume that the cache miss latency is $MissLatency = 50$ clock cycles. Note that this simplified model, assuming only one level of cache and a fixed cache miss latency, is widely used in prefetching literature; accurately modeling the cache memory hierarchy in the compiler is often too complex to be viable. Moreover, since the cache is usually a system-wide shared resource, it is impossible to model interference from external sources such as other running processes.

Mowry's widely used loop prefetching algorithm [13] starts by peeling and unrolling the loop to filter out unnecessary prefetch instructions and reduce the instruction overheads. After this step, the resulting code for our running example looks like the code in Figure 6(b).

Next, Mowry's algorithm computes the *PrefDistance* using Equation 1.

$$PrefDistance = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil \tag{1}$$

Prefetch instructions are inserted in the loop *PrefDistance* iterations before their use. It does not take into consideration the number of in-flight memory requests supported by the hardware. The maximum number of prefetch requests active at any time can be computed using:

$$MaxRequests = NumRefs \times PrefDistance \tag{2}$$

where *NumRefs* represents the number of static references that require prefetching. Going back to the code in Figure 6(c), $NumRefs = 3$ since the references to `A[i]`, `B[i]` and `C[i]` will cause a cache miss at each iteration and need prefetching, leading to $MaxRequests = 3 \times 3 = 9$. Suppose that our architecture has 6

(a)

```
for (i=0;i<1000;i++)
  A[i] = B[i] + C[i];
```

(b)

```
for (i=0;i<994;i += 2 ) { /* Unrolled */
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
/* Last three iterations peeled */
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(c)

```
for (i=0;i<994;i += 2 ) {
  /* prefetch 3 iterations in advance */
  prefetch(A[i+6]);
  prefetch(B[i+6]);
  prefetch(C[i+6]);
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(d)

```
for (i=0;i<994;i += 2 ) {
  prefetch(A[i+6]);
  prefetch(B[i+6]);
  /* Does not prefetch C */
  A[i] = B[i] + C[i] ;
  A[i+1] = B[i+1] + C[i+1];
}
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(e)

```
for (i=0;i<996;i += 2 ) {
  /* prefetch 2 iterations in advance */
  prefetch(A[i+4]);
  prefetch(B[i+4]);
  prefetch(C[i+4]);
  A[i] = B[i] + C[i] ;
  A[i+1] = B[i+1] + C[i+1];
}
/* Last two iterations peeled */
for (i=996;i<1000;i++)
  A[i] = B[i] + C[i];
```

Figure 6: (a) Original code before loop prefetching (b) Loop unrolling and peeling to isolate likely cache misses (c) Code after Mowry's prefetching algorithm (*PrefDistance* = 3) (d) Code after applying GCC loop prefetching algorithm (prefetch slots=6) (e) Outcome of the RAP algorithm: *PrefDistance* lowered to 2.

registers in the *Miss Information/Status Holding Register* (MSHR) file. After the first six prefetch requests have been issued, when the next request arrives at the MHA unit, one of the following can happen, depending on the hardware implementation:

1. The additional request is silently dropped, and nothing is sent to the lower levels of the memory hierarchy. This causes the program to slow down, since it incurs all the instruction overheads of prefetching, but none of the benefits – the cache miss was not avoided.

2. The MHA does not accept the prefetch request, stalling the issuing CPU until one MSHR becomes available (which happens when one request returns from DRAM or lower cache level). Stalling the CPU was exactly what prefetching was aiming to avoid, and thus the benefits of prefetching are again lost, leaving only the overheads.

To summarize, overflowing the MSHR file is detrimental in all cases, and needs to be addressed by all prefetching approaches. The code in Figure 6(c), which is the outcome of Mowry's algorithm, fails to address this issue. We discuss proposed improvements next.

GCC (GNU Compiler Collection), a state-of-the art open source compiler which supports a wide range of architectures and programming languages, includes an implementation of Mowry's algorithm for loop prefetching. The GCC algorithm extends it further by introducing the notion of a platform-specific number of *PrefetchSlots*. This is used to limit the number of prefetches that can be in flight at the same time. As far as we know, GCC's method is the only software prefetching algorithm that attempts to limit the number of in-flight prefetches based on hardware limitations. After performing the same steps 1-2 as above, the GCC algorithm starts scheduling prefetches for all the references in program order. One prefetch instruction issued *PrefDistance* iterations in advance of the reference causes the number of available prefetch slots to be decremented by *PrefDistance*. Once not enough *PrefetchSlots* are left, it stops issuing prefetches for the remaining references.

For our running example, Figure 6(d) shows the outcome of the GCC algorithm. Since the prefetch instructions for the `A[i]` and `B[i]` references use up all 6 available prefetch slots, no prefetch is issued for the `C[i]` reference. At runtime, this means a cache miss penalty will be encountered every iteration of the unrolled loop, significantly affecting its running time. Although the GCC algorithm in Figure 6(d) addressed the MSHR file overflowing issue encountered by Mowry's original approach in Figure 6(c), it comes short of the main goal of hiding the memory latency of all the memory references.

Figure 6(e) shows the outcome of the RAP algorithm applied to our example code. The prefetch distance has been lowered to two iterations, which allowed prefetches to be issued for all three references. As will be discussed below, with this transformation there will be only *one cache miss per three iterations*: once a cache miss is encountered, it gives enough time for all previously issued prefetch requests to complete, including current and next two iterations. By contrast, the GCC implementation encounters one miss per each iteration, which translates to three times more time spent in memory stalls.

To formulate an algorithm for RAP, it is useful to understand the limitations of GCC's prefetcher. There is a subtle inconsistency in the way GCC schedules prefetching instructions: on one hand, the prefetch distance is computed assuming all memory latencies can be hidden through prefetching; on the other hand, under certain conditions, prefetch instructions for some references are not even issued, causing some references to be cache misses. This affects the iteration time, and therefore the prefetch distance should be adjusted accordingly: if each iteration takes longer, then prefetches can be issued fewer iterations in advance and still be able to hide the latency. However, GCC does not adjust the prefetch distance in these cases, *effectively using a flawed model for scheduling prefetches.*

Figure 6(d) shows an example of the suboptimal scheduling algorithm described above. To help understand the runtime behavior, we show the resulting dynamic cache trace in Figure 7(a). The first three iterations are not prefetched for, hence all references are cache misses. At each iteration from $i = 6$ onward, the read from `C[i]` is going to be a cache miss, which on our hypothetical architecture takes 50 clock cycles. This is 49 cycles more than in the original estimate, and thus $IterTime = 20 + 49 = 69$. Using Equation (1), we need $PrefDistance = \lceil 50/69 \rceil = 1$ iteration in advance. However, GCC schedules prefetches using $PrefDistance = 3$ iterations in advance, according to the original calculation. Moreover, because of this

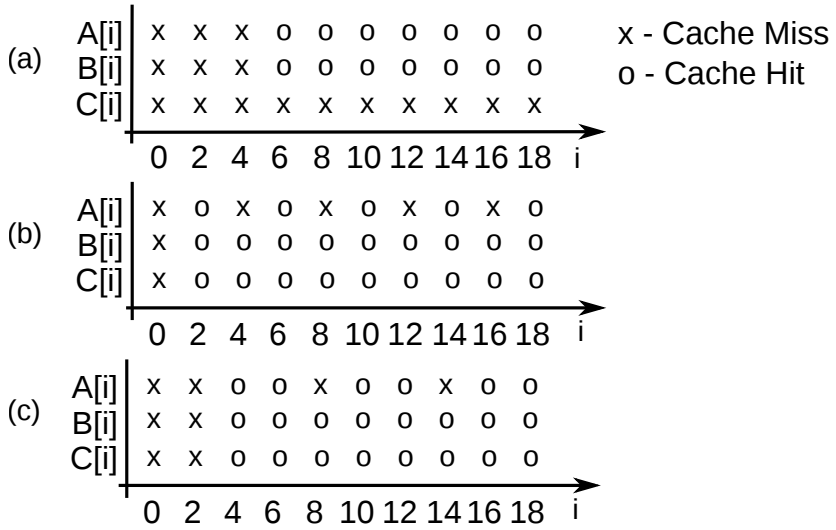

Figure 7: Dynamic cache trace for the code in Figure 6. (a) GCC loop prefetching with $PrefDistance = 3$ and (b) RAP algorithm with $PrefDistance = 1$ (c) RAP algorithm with $PrefDistance = 2$.

inconsistency, no prefetch instruction is inserted for the `C[i]` reference, causing a miss at every iteration as illustrated in Figure 7(a).

Let us examine an alternative scheduling algorithm in which a smaller *PrefetchDistance* is used. The RAP algorithm discussed in the rest of this section is based on this scheme. If we use $PrefDistance = 1$ iteration instead of 3, we can now issue prefetches for all three references, using a total of $MaxRequests = 3 \times 1 = 3$ prefetch slots. The cache trace for this case is shown in Figure 7(b). When $i = 0$, we issue prefetch requests for `A[2]`, `B[2]` and `C[2]`, then we encounter three cache misses for `A[0]`, `B[0]` and `C[0]`. For $i = 2$, we start by issuing prefetches for iteration $i + 2 = 4$, then all references are cache hits, because the prefetch requests issued at the beginning of iteration $i = 0$ overlapped with the previous misses and have had time to complete (see Figure 7(b)). For $i = 4$, we have a cache miss for `A[4]`, but that gives enough time for the prefetches for `B[4]` and `C[4]` to complete, and thus they become cache hits. The cache miss for `A[4]` also gave enough time for all prefetches for iteration $i = 6$ to complete, meaning we have three cache hits in that iteration. The execution enters a *steady state* at this point, with one cache miss every other iteration, until the end of the loop.

Similarly, we can also use $PrefDistance = 2$, which yields the code in Figure 6(e) and the trace in Figure 7(c). Following a similar reasoning, we observe that in the steady state we encounter one miss every 3 iterations, leading to:

Claim 1 *Let* $PD_{Mowry} = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil$ *the prefetch distance computed by Mowry's algorithm (and also GCC). For any prefetch distance* $PD_{RAP} < PD_{Mowry}$ *and*

$$PD_{RAP} \times NumRefs \leq PrefetchSlots \tag{3}$$

we can issue prefetch instructions PD_{RAP} *iterations in advance for all references without exceeding the available PrefetchSlots (number of MSHR entries), and this will result in exactly one cache miss per* $PD_{RAP}+1$ *iterations in the steady state.*

The claim can be easily verified: once a cache miss has been encountered, it allows enough time for all the prefetch requests already issued for the next PD_{RAP} iterations to complete, ensuring they are all hits. However, since PD_{RAP} iterations with all hits do not provide enough time to hide the miss latency, iteration $PD_{RAP} + 1$ encounters a cache miss for the first read. The cycle then repeats.

Using Claim 1, we can compute the average loop iteration time in the steady state when $PD_{RAP} <$

PD_{Mowry}:

$$AvgIterTime = IterHit + \frac{IterMiss - IterHit}{PD_{RAP} + 1} \quad (4)$$

where *IterHit* is the iteration time when all references are hits (20 cycles in our example) and *IterMiss* is the iteration time with one cache miss (69 for our example).

The average iteration time (4) is a strictly decreasing function of the prefetch distance PD_{RAP}. To minimize the overall execution time, we use the upper bound value:

$$PD_{RAP} = \left\lfloor \frac{PrefetchSlots}{NumRefs} \right\rfloor \quad (5)$$

given by (3). In the example in Figure 6(e), we have $PD_{RAP} = \lfloor 6/3 \rfloor = 2$. We can now present our improved compiler algorithm:

Algorithm 1 Resource-Aware Prefetching

I-II. *Identical to Steps* ***I-II*** *in Algorithm 1.*

III. *Compute* $PD_{Mowry} = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil$ *and NumRef the number of references. Let* $PD_{RAP} = \left\lfloor \frac{PrefetchSlots}{NumRefs} \right\rfloor$.

- ***III.1*** *If* $PD_{Mowry} \times NumRefs \leq PrefetchSlots$*, schedule prefetch instructions for all NumRef references* PD_{Mowry} *iterations in advance.*
- ***III.2*** *If* $PD_{Mowry} \times NumRefs > PrefetchSlots$ *and* $PD_{RAP} \geq 1$*, schedule prefetch instructions for all NumRef references* PD_{RAP} *iterations in advance.*
- ***III.3*** *If* $PD_{Mowry} \times NumRefs > PrefetchSlots$ *and* $PD_{RAP} = 0$*, schedule prefetch instructions for the first PrefetchSlots references in program order exactly* ***one*** *iteration in advance.*

Case III.1 corresponds to the non-resource restricted situation, where we fall back on the same scheduling algorithm as Mowry's (and GCC) algorithm. Case III.2 occurs in situations when there are not enough *PrefetchSlots* to completely hide all cache misses; the algorithm issues one prefetch for each reference using a smaller prefetch distance, resulting in one cache miss every PD_{RAP+1} iterations. Case III.3 occurs in severely resource-constrained cases, where we have more static references than *PrefetchSlots*. The algorithm issues prefetch instructions one iteration ahead to as many references as possible, without exceeding *PrefetchSlots*.

4.1.1 Prefetching after reference

In most modern architectures, a MSHR entry is allocated for a prefetch request as soon as it is issued, affecting the value of the *MaxRequests* value. Consider a memory reference `A[i]` and its associated prefetch instruction `prefetch(A[i+PD])`, with PD the prefetch distance. Right before `A[i]` is accessed we will have pending (unconsumed) prefetches for references `A[i]`, `A[i+1]`,... `A[i+PD-1]`, using up a total of PD MSHR entries. At this point we can chose to issue the prefetch for `A[i+PD]` before or after the reference to `A[i]`. If the prefetch is inserted textually in the code before the reference to `A[i]` (i.e. as in Figure 6), an additional MSHR will be needed (for a total of $PD + 1$). Alternatively, the prefetch instruction for `A[i+PD]` can be inserted in the code right *after* the reference to `A[i]`. Hence one prefetched value will be consumed before issuing the new one, thus require reserving only PD MSHR entries. This leads to a saving of one MSHR entry per reference prefetched, at the cost of hiding slightly less latency, since the prefetch is now issued closer to the use. In severely resource-constrained environment, this minor optimization can have a non-trivial effect on performance.

For the implementation of the RAP algorithm, we used the "prefetch-after reference" optimization.

4.1.2 Implementation in the XMTC Compiler

We implemented the **resource-aware loop prefetching algorithm** (RAP) as an optimization pass within the GCC compiler. It operates using the TreeSSA framework [15] which was introduced starting with GCC 4.x releases.

The RAP algorithm operates on the code executed by each TCU while in parallel mode. It optimizes only the inner-most loops of the code, leaving outer loops or non-loop code unchanged. The pass inserts prefetch instructions that bring data from lower levels of the memory hierarchy (shared cache or DRAM) into the prefetch buffers located at each TCU. To implement the RAP algorithm, we used some of the analyses that are part of GCC:

- *Loop induction variable analysis:* Identify loop induction variables in the code. Also used to express the address of a memory reference as an affine function in the loop induction variable if possible, identifying the base and stride.
- *Estimate the duration of a loop iteration:* Using "weights" for each type of instruction to estimate the number of cycles needed for a loop iteration. This is needed to compute the prefetch distance.

Two important parameters of the RAP algorithm are the size of the MSHR file and the latency to the shared cache. These can be read from a architecture-specific configuration file provided with the compiler, or can be specified by the user by passing specific command line arguments to the compilation process.

The RAP pass can be enabled or disabled by the programmer using command-line arguments. We have not observed any slowdowns caused by enabling the prefetching pass, but, as with any optimization pass, we recommend to the performance programmer to experiment with and without this optimization to chose the best performing configuration during the testing and optimization phases of development.

4.2 Burst Prefetching

Burst prefetching hides latency when more than one contiguous memory load instruction is encountered. This is common in RISC architectures such as XMT, especially when a function returns, and the callee has to restore the *callee saved* registers it modified and the caller must restore all *caller saved* registers. Load operations are blocking, so a sequence of N loads requires N round-trips to memory, but this number can be reduced by means of prefetching. Linear prefetching misses this opportunity because it is implemented in the front-end of the compiler, before register allocation and stack management code generation. Conversely, burst prefetching is implemented in the post-pass with the specific goal to act after those compiler passes produce assembly code. Like linear prefetching, burst prefetching is useful because of the lack of local caches that would provide spatial locality.

(a) Assembly Produced by GCC	**(b) After Burst Prefetching**
	`pref 8, 44($sp)` *// 1*
	`pref 8, 40($sp)` *// 1*
	`pref 8, 36($sp)` *// 1*
	`pref 8, 32($sp)` *// 1*
`lw $31, 48($sp)`	`lw $31, 48($sp)` *// RTTM*
`lw $23, 44($sp)`	`lw $23, 44($sp)` *// 1*
`lw $22, 40($sp)`	`lw $22, 40($sp)` *// 1*
`lw $21, 36($sp)`	`lw $21, 36($sp)` *// 1*
`lw $20, 32($sp)`	`lw $20, 32($sp)` *// 1*
5 Round-Trips to Memory(RTTM)	**1 RTTM + 8**

Figure 8: Simple Burst Prefetching Example.

Figure 9(a) shows a snippet of the assembly code produced by the core-pass (GCC) when compiling the *QUEENS* benchmark [16]. Nine consecutive load instructions are issued to restore registers before returning

(a) Assembly Produced by GCC	(b) After Burst Prefetching
	`pref 8, 44($sp)  // 1`
	`pref 8, 40($sp)  // 1`
	`pref 8, 36($sp)  // 1`
	`pref 8, 32($sp)  // 1`
`lw $31, 48($sp)`	`lw $31, 48($sp)  // RTTM`
`lw $23, 44($sp)`	`lw $23, 44($sp)  // 1`
	`pref 8, 28($sp)  // 1`
`lw $22, 40($sp)`	`lw $22, 40($sp)  // 1`
	`pref 8, 24($sp)  // 1`
`lw $21, 36($sp)`	`lw $21, 36($sp)  // 1`
	`pref 8, 20($sp)  // 1`
`lw $20, 32($sp)`	`lw $20, 32($sp)  // 1`
	`pref 8, 16($sp)  // 1`
`lw $19, 28($sp)`	`lw $19, 28($sp)  // RTTM-6`
`lw $18, 24($sp)`	`lw $18, 24($sp)  // 1`
`lw $17, 20($sp)`	`lw $17, 20($sp)  // 1`
`lw $16, 16($sp)`	`lw $16, 16($sp)  // 1`
9 Round-Trips to Memory (RTTM)	**2 RTTM + 9**

Figure 9: Complete Burst Prefetching Example.

from a function, which results in nine round-trips to memory (RTTM). Figure 9(b) shows the same code after the burst prefetching has inserted prefetching instructions. Burst prefetching is *resource-aware*, taking the size of the prefetch buffer as an input, but it does not try to hoist the inserted prefetches after they are inserted. The key idea is to use a prefetch-buffer miss (similar to a cache miss) to hide the latency of multiple prefetches.

To illustrate that concept we use a simpler example. Figure 8(a) shows five consecutive load instructions resulting in five RTTM. Assuming the prefetch buffer can hold four words (elements), burst prefetching will issue prefetch instructions for the last four load instructions as shown in Figure 8(b). The first load instruction will incur a round trip to memory since it was not prefetched (prefetch-buffer miss), but during that time, the four issued prefetches will presumably have time to complete. In reality, queuing at various points in the system may cause them not to complete, but they will have made substantial progress nonetheless. Overall, the code after burst prefetching takes 1 RTTM + 8 cycles to complete, as opposed to 5 RTTM for the original code. This is substantial, considering that one RTTM takes around 25 cycles on the 64 TCU XMT FPGA and can take longer on XMT configurations with more TCUs. Note that the burst prefetching will issue prefetches for all the load instructions if they fit in the prefetch buffer: if there were 4 loads in the above example, they would all be prefetched.

The example in Figure 8 illustrates that with a prefetch buffer of K words, the number of RTTMs can be reduced by a factor of $K + 1$ and a small number of cycles will be added for issuing the prefetching instructions. Figure 9 shows what happens when the number of consecutive load instructions is larger than K (but smaller than $2K$). The first part is identical: four prefetches are issued; then, load instructions are inserted and as prefetch buffer locations are freed by them, more prefetches are issued.

Two rules form the core of the burst prefetching algorithm as it traverses the list of consecutive loads. First, if the number of remaining load instructions is greater than the size of the prefetch buffer, a load is *skipped* (no prefetch is issued for it) and the miss it will cause will allow the pending prefetches to complete. Second, prefetches are issued as soon as possible taking into account the size of the prefetch buffer. When pending prefetches fill up the prefetch buffer, load instructions are pushed to the output instruction list until a prefetch buffer location is freed (consumed).

Algorithm 2 gives a verbal description of the algorithm of burst prefetching and Algorithm 3 presents its pseudocode.

Algorithm 2 Burst Prefetching Algorithm

INPUT: *instructions*: list of load instructions
OUPTUT: *output*: list of instructions including prefetching
while you have not considered all instructions for prefetching **do**
▷ Go through the *instructions* list from head to tail
  if The number of load instructions remaining is strictly larger than the size of the Prefetch Buffer AND there is no upcoming load instruction that was not prefetched (that was skipped) **then**
    Skip prefetching for the load instruction under consideration, keep track of the instruction (and consider the next load instruction in the next iteration)
  else
    Build a prefetch instruction for the load currently under consideration and add it to the *output*. Also increment the number pending prefetch instructions.
    while The number of pending prefetches is $\geq$ than the Prefetch Buffer Size **do**
      Move a load instruction from the head of *instructions* to the tail of *output*
      If that load instruction was prefetched, decrement the number of pending prefetches, else (it was skipped) mark that there is no longer an upcoming load instruction that was not prefetched.
    end while
  end if
end while
Enqueue the rest of *instructions* to *output*

Closed Formula for Burst Prefetching. To define the closed form formula for the number of cycles taken by a sequence of lw instructions given a prefetch buffer of size S, we first define some values. Let L_{RTTM} be the number of load instructions that will require a round trip to memory after burst prefetching. Those are $L_{RTTM} = \lceil \frac{lw}{S+1} \rceil$. The number of remaining load instructions is $L_R = lw - L_{RTTM}$. The number of load instructions that were **not** prefetched is $N = \lfloor \frac{lw}{S+1} \rfloor$, and the number of prefetched instructions is $P = lw - N$. Let O be the number of overlapped cycles by prefetches issued before a load but without an intermediate RTTM operation. This number is convoluted and only reduces the cycle count slightly. Without it (assuming it is zero), we still get a pretty accurate upper bound on the number of cycles it takes to execute lw load instructions with burst prefetching, using a prefetch buffer of size S (see Equation 6).

$$Cycles(lw, S) = RTTM \cdot L_{RTTM} + L_R + P - O \tag{6}$$

To define the number of overlapped cycles O we need to define the following values: let $m = lw \bmod (S+1)$; let a be zero if $m = 0$ and one otherwise ($a = \{0 \text{ if } m = 0, 1 \text{ if } m \neq 0\}$); let c be zero if $lw \leq S$ and one otherwise ($c = \{0 \text{ if } lw \leq S, 1 \text{ if } lw > S\}$). The number of overlapped cycles is then:

$$O = a \cdot ((m-1) + c \cdot (S-1)) \tag{7}$$

4.3 Prefetching Compatibility

The three types of prefetching (linear, loop, and burst prefetching) are not yet aware of each other. This means that when more than one of the three prefetching passes are enabled there is a distinct probability that more prefetches will be issued than there are prefetch buffers, resulting in thrashing and in loss of performance. Ideally, linear prefetching and burst prefetching can be performed in a signle back-end pass by adding a resource-aware instruction scheduling pass them to hoist the prefetch instructions. Furthermore, to make this combined back-end prefetching aware of the front-end loop prefetching, we would need a *forward may* analysis identifying how many prefetch buffers are in use at each node of the control-flow graph (CFG).

Algorithm 3 Burst Prefetching Pseudocode

```
1: procedure BurstPrefetching(instructions)                    ▷ List of load instructions
2:     hasSkipped ← false                                      ▷ true when the next load was not prefetched
3:     prefCnt ← 0                                             ▷ Number of pending prefetches
4:     read ← instructions.head                                ▷ Pointer to read next load
5:     insert ← instructions.head                              ▷ Pointer to insert next load
6:     ouput ← ∅                                               ▷ List of instructions with prefetches
7:                                                             ▷ Invariant A: insert(.next)^(prefCnt+s) = read,
8:                                                             ▷ where s = 1 if hasSkipped = true, and s = 0 otherwise.
9:     while read ≠ instructions.tail do
10:        if hasSkipped = false ∧ remaining loads > PrefBuff.size then
11:                                                            ▷ Skip load instruction pointed to by read
12:            hasSkipped ← true
13:            skipped ← read
14:            read ← read.next
15:        else
16:            pref ← build prefetch instruction for read
17:            output.enqueue(pref)
18:            read ← read.next
19:            prefCnt ← prefCnt + 1
20:            while prefCnt ≥ PrefBuff.size do                ▷ Invariant B: prefCnt ≤ PrefBuff.size
21:                output.enqueue(insert)
22:                insert ← insert.next
23:                if hasSkipped = true ∧ skipped = insert then
24:                    hasSkipped ← false                      ▷ Consumed a load that was not prefetched
25:                else
26:                    prefCnt ← prefCnt − 1                   ▷ Consumed a prefetched load
27:                end if
28:            end while
29:        end if
30:    end while                                               ▷ Invariant C: read = tail
31:    while insert ≠ tail do          ▷ Flush remaining load instruction (Invariant A stops holding)
32:        output.enque(insert)
33:        insert ← insert.next
34:    end while
35:    return output
36: end procedure
```

5 Compiling a flat XMTC spawn statement

This section focuses on how a flat spawn statement, i.e., one without nested parallelism, is compiled down to assembly. The interesting aspect is how the XMT hardware is harnessed for scheduling such outer spawn statements. For simplicity, we show how outer spawns were compiled before nested parallelism was supported, and later (Sections 6.7 and 6.6), we revisit this information to include support for nested parallelism.

Figure 10(a) shows a generic spawn statement. Its arguments, low and high, are expressions that evaluate to integers. The `BlockCode` is parametric in the task ID ($). In Figure 10(b), the core-pass (GCC) generates high level XMTC assembly: the `spawn` instruction has two integer register arguments that hold the values of the lowest and the highest task IDs to be executed; the `join` instruction marks the end of the task code; the `BlockAssembly` is a straightforward compilation of the `BlockCode`, before which the live Master TCU registers are broadcast to the corresponding TCU registers. The `spawn` and `join` instructions are further

(a) XMTC code	(b) after core-pass	(c) after post-pass
		1 mvtg $grHigh, $rHigh 2 getid $rTmp, $rLow 3 mvtg $grLow, $rTmp 4 **spawn** 5 broadcast $rLow, $rLow 6 getid $rId, $rLow

spawn(low, high)
{
BlockCode(**$**)
}
	spawn $rLow, $rHigh
	bcast live regs
	BlockAssembly(**$**)
	join
		7 spawn_start:
		8 chkid $rId, $grHigh
		9
		10 bcast live regs
		11 BlockAssembly(**$**)
		12
		13 move $rId, 1
		14 **ps** $rId, $grLow
		15 jump spawn_start
		16 **join**

Figure 10: Compiling a flat spawn statement

expanded by the post-pass to make use of XMT's capabilities of hardware scheduling and synchronization as shown in Figure 10(c).

The XMT specific instructions used in Figure 10(c) to expand the `spawn` and `join` statements are explained below. Remember that each TCU and the Master TCU have their own private set of local registers and that global registers can be accessed and modified both by the Master TCU and by all parallel TCUs. Local registers are named $rX and global registers as $grX.

- **getid $rX, $rY:** When used in serial mode (i.e., by the Master TCU), it adds the value of register $rY to the number of TCUs in the system and stores the result in register $rX (i.e., $rX ← $rY + nTCUs). When used in parallel mode, it adds the value of register $rY to the identification number of the TCU executing the instruction and stores the result in register $rX (i.e., $rX ← $rY + TCU-ID). TCUs are numbered from zero to $P-1$ on a P-TCU XMT.

- **mvtg $grX, $rY:** Moves the value of register $rY to global register $grX (i.e., $grX ← $rY). This instruction is only valid in sequential mode (it can only be executed by the Master TCU).

- **broadcast $rX, $rY:** Copies the value of register $rY of the Master TCU to register $rX of the TCU executing the instruction (i.e., $rX[TCU] ← $rY[MTCU]). This instruction is only valid in parallel mode (it can only be executed by TCUs but not the Master TCU).

- **ps $rX, $grY:** Performs the hardware prefix-sum operation. It atomically adds the value of register $rX to global register $grY and sets $rX to the value of $grY before the operation (`atomic`{tmp←$grY; $grY←$grY+$rX; $rX←tmp}). This instruction is only valid in parallel mode (it can only be executed by TCUs but not the Master TCU).

- **chkid $rX, $grY:** This instruction is only valid in parallel mode (it can only be executed by TCUs but not the Master TCU). It compares the value of register $rX to the value of global register $grY. If $\$rX \leq \$grY$ execution proceeds to the next instruction; otherwise the TCU blocks until $\$rX \leq \$grY$, e.g., some other TCU increments $grY, or until all TCUs are blocked at a `chkid` instruction, signaling the end of the parallel section and the return to sequential execution on the Master TCU (i.e., `while`($rX[TCU] > $grY){TCU is blocked and if all TCUs are blocked switch to MTCU}). This instruction implements a type of barrier and allows quick and efficient transition from parallel to sequential execution.

The expansion of the high level `spawn` and `join` instructions are shown in Lines 1-8 and 13-16 of Figure 10(c). Line 1 sets $grHigh to $rHigh, the ID of the last task to execute. Line 2 sets $rTmp to the sum of the number of TCUs and $rLow, the ID of the first unallocated task to execute ($\$rTemp = low + |TCUs|$) and Line 3 sets the global register $grLow to the value of $rTemp, so that $grLow will hold the ID of the next available task. The number of TCUs is added to *low* because each TCU will automatically initialize its task ID to $Task_{id} = low + TCU_{id}$ (Line 6), as soon as control is transfered to them with the `spawn` instruction (Line 4). Each time a TCU completes its task, it will atomically increment $grLow using a prefix sum operation (Line 14) and check if the task ID is valid, i.e., $\leq \$grHigh$ (Line 8). Parallel execution starts at Line 4 and the value of $rLow is broadcast from the MTCU to the TCUs (Line 5) as it will be needed for the TCUs to compute their initial task IDs (Line 6). Line 8 only allows TCUs with valid IDs to proceed while blocking all the others. The expansion of the join is straightforward: Line 13 sets an increment of 1 for the prefix-sum operation of Line 14, which returns the next available task-ID (and increments $grLow); then, the execution jumps to the `checkid` instruction (Line 8) to validate this newly acquired task-ID.

XMT's hardware implements a first-in first-out (FIFO) schedule of the tasks, akin to using a global queue for scheduling. While this is good for flat parallelism, it results in a potentially unbounded memory footprint in the presence of nested parallelism. An interesting question is how to best use XMT's hardware to support nested parallelism, while keeping the memory footprint bounded. As mentioned previously, we use the hardware to schedule outer spawn statements and use software scheduling for nested spawn statements. More details on how this is achieved are provided in Section 6.7.

6 Nested Parallelism Support

Since we want to use XMT's hardware to schedule outer spawn statements, but we want to fall back on software work-stealing scheduling for nested spawns to avoid the unbounded memory footprint of global queue scheduling, we need a way to distinguish outer spawns from nested ones. For example, although the `spawn` on Line 3 is certainly nested in Figure 11, we do not know for sure whether the one on Line 2 is nested or not. It depends on whether the function `foo` was called from a sequential or a parallel context (execution mode). To enable this distinction, we perform function cloning presented in the next section.

```
1  void foo() {
2    spawn (1, 100) { // Outer?
3      spawn (1, 50} { // Nested
4            ...
5      }
6  }
```

Figure 11: Identifying Outer and Nested Spawns

6.1 Function Cloning

Outer spawn statements can be identified statically after performing *function cloning*. A parallel clone of each function is created and all function call-sites updated to call the appropriate clone. The original function is called from sequential contexts, and the parallel clone is called from parallel contexts (Figure 12).

Function pointers are not currently supported in XMTC. To support them, the appropriate clone would have to be selected dynamically at run-time, unless the compiler could statically disambiguate which clone was needed. The mechanism for picking the right clone at run-time is similar to picking the appropriate version of a *virtual* function of an object based on its type in object oriented languages. We do not expect this to be difficult to resolve in the future as the XMT platform matures.

(a) original code

```
void foo( ... ) {
  bar();    // function call
  spawn (low, high) {
    bar(); // function call
  }
}
```

(b) after function cloning

```
void foo( ... ) {
  bar();        // function call
  spawn (low, high) {
    par_bar(); // function call
  }
}
void par_foo( ... ) {
  par_bar();    // function call
  spawn (low, high) {
    par_bar(); // function call
  }
}
```

Figure 12: Function Cloning

6.2 Function Call Support in Parallel Code: Stack Allocation

To support recursively nested parallelism, the programmer must be allowed to call a function from parallel code. To make this possible, we implemented parallel stack allocation. However, parallel stack allocation is trickier than simply maintaining a linear stack for each TCU. The reason is that according to the semantics of a spawn statement, its spawn block has access to the variables in the enclosing scope. For example, in Figure 13(a) the parameter N should be accessible within the spawn block, but because it is modified, a single copy of N is maintained. Multiple copies may be preferable in the case of variables that are only read by tasks. Figure 13(b) shows that tasks should have their own activation frame as they can be running on different workers, but they should have access to the frame of the parent task, which in the example holds N. Such a tree of activation frames is called a *cactus-stack*.

(a) Example Code

```
int foo(int N) {
  spawn(1, 100) {
    atomic{ N += $ };
  }
  return N;
}
```

(b) Cactus-Stack

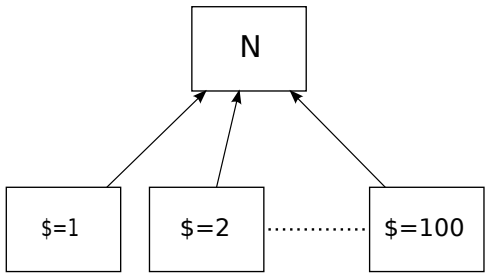

Figure 13: Spawn Scope Example

A cactus-stack can be implemented many different ways. An obvious way is to dynamically allocate each activation frame and keep a pointer to the parent frame. But when a task calls a function sequentially, this overhead is unnecessary, and something more akin to a linear stack could be used.

In the fourth chapter of his PhD thesis [7], Goldstein presents and compares four implementations of cactus-stacks: linked frames, spaghetti stacks, stacklets, and multiple stacks. The trade-offs of these implementations include allocation efficiency, internal fragmentation, external fragmentation, and ease of implementation. Internal fragmentation is caused by leaving unused space within the basic allocation unit and external fragmentation arises when allocating activation frames in different regions of the memory space.

Of the four alternatives, *stacklets*, proposed by Goldstein in his earlier work [8], achieved the best performance, especially for fine-grained parallelism. We chose to implement a cactus-stack based on Goldstein's *stacklets* on XMT, despite the fact that implementing them is relatively complex. Having a sequential and a parallel clone of each function minimizes stack allocation overheads for sequential parts of the code that

do not need the additional complexity of cactus-stacks. This can also be achieved by compiling functions to have two entry points in assembly, one which includes stacklet allocation and the other skipping it. In any case, stack is allocated as usual for the sequential clone of a function, and the more complex stacklet allocation code of the cactus-stack is only generated for the parallel clone.

The main motivation of stacklets is to reduce the stack allocation cost in the common event of a task calling a function sequentially, as opposed to spawning new tasks. In that case, the allocation will be almost as efficient as sequential stack allocation.

A stacklet is a continuous chunk of memory (2KB in the XMTC implementation) with some necessary information, such as a link to the parent stacklet. Activation frames are allocated within a stacklet as on a sequential stack, with just one additional check that the activation frame to be allocated fits in the remaining space in the stacklet; otherwise a new stacklet is allocated, linked to the current stacklet and the activation frame is allocated at the base of the new stacklet.

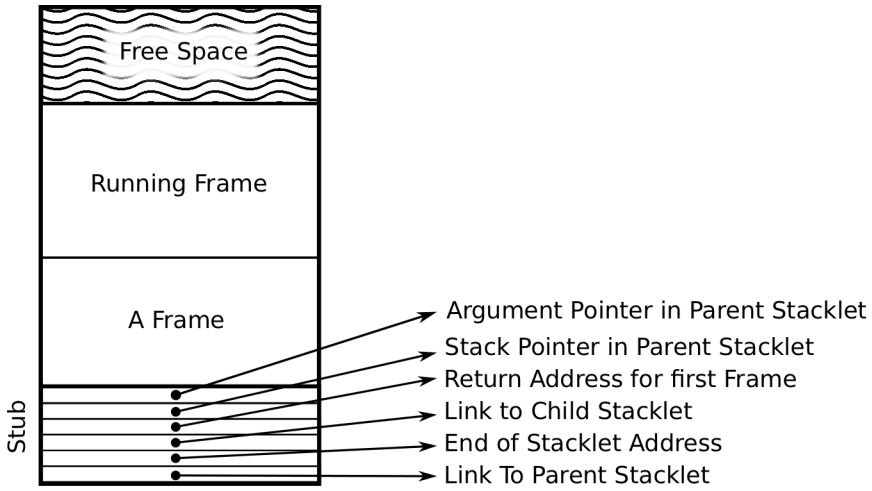

Figure 14: Stacklet

Figure 14 shows a stacklet. It has a stub, some activation frames and free space at its top. The activation frame at the top of the stacklet is the running frame, unless the stacklet has an active child. The stub stores some necessary information to keep stacklets linked together and to restore the state of a function returning to the parent stacklet, such as the return address, the stack pointer and the arguments pointer (`argp`).

The arguments pointer is needed because the arguments of a function and its activation record may not be contiguous. In sequential execution, the running frame and the parent frame are (usually[3]) contiguous in memory. The caller pushes the arguments of the function it calls onto its activation frame (parent frame) and calls the function, which accesses its arguments through its stack pointer because they lie right above its activation frame. On the contrary, in the stacklet implementation of the cactus stack, the parent frame may not be immediately above the running frame, for example in the case of a stacklet overflow when a new stacklet is allocated. The arguments pointer gives a solution to this problem by keeping a separate pointer to the function's arguments. Just like for the stack pointer, a register is reserved for the arguments pointer.

A worker thread needs to have access to the stacklet stub when checking if a new allocation frame will fit in the stacklet. The frame-pointer register (`$fp`) is employed for that purpose, since it is not used for a language like C. For a language with nested functions, however, the frame pointer would be used, and a different register would have to be reserved to point to the stacklet stub.

[3]This may not be true for languages that support nested function declarations, but XMTC does not support them.

Algorithm 4 Function Prologue Expansion for supporting Stacklets

```
1: if # [Frame_Size = 0 && callee is a leaf function] then
2:     Sequential Function Prologue Expansion
3:     return
4: end if
5: Save Arguments pointer $argp in current activation frame
6: $argp ← $sp                          ▷ Keep a pointer to the arguments before updating $sp
7: if #flag [align_cactus_stack] then
8:     EndOfStacklet = $sp && STACKLET_MASK
9: else
10:    EndOfStacklet = $fp.end_address                    ▷ $fp points to base of stacklet
11: end if
12: EndOfNewFrame = $sp - frame_size                      ▷ Stack grows downward
13: if EndOfNewFrame < EndOfStacklet then                 ▷ Frame doesn't Fit in Stacklet
14:    $fp.saved_sp ← $sp
15:    $fp.saved_argp ← $argp
16:    if $fp.child_stacklet = NULL then                  ▷ Allocate child stacklet
17:        Decrement ← − STACKLET_SIZE                    ▷ STACKLET_SIZE = 2K
18:        NewStacklet ← psm(Decrement, Global_SP)
19:        $fp.child_stacklet ← NewStacklet
20:        NewStacklet.child_stacklet ← NULL
21:        if #flag [NOT align_cactus_stack] then
22:            NewStacklet.end_address ← NewStacklet - STACKLET_SIZE
23:        end if
24:        $fp ← NewStacklet
25:    else
26:        $fp ← $fp.child_stacklet
27:    end if
28:    $sp = $fp - STUB_SIZE
29:    $fp.return_address ← $return_register
30:    $return_register ← deallocate_stacklet
31: end if
32: Sequential Function Prologue Expansion + allocate space for saving and restoring $argp
```

Algorithm 4 shows how a function prologue is expanded to support stacklets, using sequential function prologue expansion as a subroutine. The sequential expansion involves saving and restoring the callee-saved registers, and updating the stack pointer. The expansion of Algorithm 4 is only applied to the parallel clone of a function, leaving the sequential clone optimized for sequential execution. Keep in mind that the stack grows downward, by reducing a global stack pointer (Global_SP) atomically (Line 18). Also, note that in Lines 1-4 the sequential expansion is used if the function does not need a frame (size=0) and does not call other functions (it is a *leaf function*). The rest of this code checks if the new activation frame needed by the function being called fits in the current stacklet (Line 13), and if not, it allocates a new stacklet off the global stack, using an atomic prefix-sum (fetch-and-add) operation on Line 18. Then, the return address of the function is saved (Line 29) and set to the stacklet deallocation routine, shown in Algorithm 5. The current implementation of the cactus stack does not support freeing and reallocating stacklets within a parallel section, so simply using the prefix-sum is a quick and simple solution for allocating stacklets.

The deallocation routine restores the TCU state registers (sp, fp, argp) and returns to the right instruction. In a production implementation, the child stacklet would possibly be returned to a pool of free stacklets. In this prototype, however, this complexity was not necessary because we are keeping stacklets linked and reusing them, as discussed later in this section.

Algorithm 5 Stacklet Deallocation

1: `$return_register` ← `$fp`.return_address
2: `$fp` ← `$fp`.parent
3: `$sp` ← `$fp`.saved_sp
4: `$argp` ← `$fp`.saved_argp
5: jump to `$return_register`

One possible optimization is to align stacklets in memory, and make them all the same size (we picked a size of 2K). That way, checking if a new frame will fit in the current stacklet (Line 13) does not require any memory access: since the size of the stacklet is a constant, the end of the stacklet can be computed by applying a mask to the stack pointer (`$sp`). The alignment of stacklets is realized by Lines 7-9 and 21-23. Note that these `if` statements control how the expansion is performed and do not appear in the generated code. In our experience, this optimization improves performance modestly (around 2%). One disadvantage of this optimization is that the size of stacklets has to be the same for all functions, including linked libraries. Moreover, if an activation frame is bigger than what a stacklet can fit (in our implementation 2K-STUB_SIZE), the compilation fails. In XMTC, this is remedied by recompiling with a flag that disables this stacklet alignment.

In retrospect, we can achieve the same effect of checking if a new activation frame will fit in the current stacklet without accessing memory by placing the stub at the end of the stacklet instead of the beginning, and having the frame-pointer set at the start of the stacklet stub. In this scenario, the check can be performed by comparing the address of the frame pointer with that of the stack pointer decremented by the size of the requested activation frame. We have not implemented this alternative.

Another optimization we added on top of Goldstein's original stacklet design is to keep track of the child stacklet owned by the same worker that owns the parent in order to avoid the need to free and reallocate it. It is based on the observation that the stack grows and recedes multiple times during parallel execution, especially with recursive codes, which would lead to the frequent allocation, freeing, and reallocation of stacklets. Instead of freeing a child stacklet upon returning, the parent stacklet keeps a pointer to it and reuses it later if needed, as shown by the decision made on Line 16 of Algorithm 4.

If a worker (i.e., a TCU) is unable to allocate a stacklet, unused stacklets can be garbage-collected (and their parent stacklets updated to point to NULL) or even stolen. These more advanced stacklet management techniques are not currently implemented, however. Once a stacklet is allocated in the current XMTC implementation, it will remain under the ownership of the worker that allocated it until the end of the parallel section where it was allocated (i.e., stacklets cannot be returned to a pool of free memory). This has not been a problem in practice because the space allocated to stacklets is reclaimed upon returning to sequential execution, which happens efficiently and frequently on XMT, and because keeping a link to child stacklets allows for good reuse of stacklets.

Algorithm 6 Function Prologue Expansion for Outer-Spawn function

1: `$argp` ← `$sp` ▷ Set the arguments pointer
2: Tmp ← `$sp` − (STACKLET_SIZE · #TCUs) ▷ Reserve one stacklet per TCU
3: **if** #flag [align_cactus_stack] **then**
4: Tmp ← Tmp && STACKLET_MASK
5: **end if**
6: Global_SP ← Tmp
7: **if** #flag [align_cactus_stack] **then**
8: `$rId` ← `$sp` ▷ Save `$sp`. Use `$rId`, unused in sequential mode
9: `$sp` ← `$sp` && STACKLET_MASK ▷ Necessary because `$sp` is broadcast
10: **end if** ▷ Right after the `join` restore the `$sp`

There are also a type of sequential function clones that need special treatment to support stacklets: functions that introduce parallelism (by means of a spawn statement). Luckily, these are easy to identify because of the *outlining* pass described in Section 3.1, which outlines outer spawn statements and places each one in a separate function. Algorithm 6 shows the expansion of the prologue of those functions. This expansion does the following: *(1)* it initializes the arguments pointer; *(2)* it allocates an initial stacklet for each TCU; *(3)* if stacklets are aligned, it saves and aligns the stack pointer register of the Master TCU, as it will be broadcast to the parallel TCUs and used as a base for them to access their own initial stacklet; and *(4)* it initializes the global stack pointer to right after the space reserved for the initial TCU stacklets. More specifically, after the arguments pointer is set (Line 1), a stacklet is reserved for each TCU (Line 2) and the global stack pointer (implemented as a regular global variable) is initialized (Line 6), after possibly aligning it (Line 4). Moreover, if the stacklets are aligned, the sequential stack pointer is saved (Line 8) and aligned (Line 9), so that it will be aligned when broadcast to the TCUs. Otherwise, the TCUs would have to perform the alignment, possibly at the beginning of each (outer) task. The stack pointer is restored upon the completion of the spawn statement, as part of the join expansion performed by the post-pass. Lines 1-6 are performed by the core-pass, whereas lines 7-10 are performed by the post-pass as part of the spawn-expansion, similarly to Figure 10[4]. Note that the expansion for such outlined functions introducing parallelism should not allocate space on the serial stack. To this end, we had to override GCC's stack allocation and make a special case for such functions.

```
1    mvtg  $grHigh, $rHigh
2    getid $rTmp, $rLow
3    mvtg  $grLow, $rTmp
4    spawn
5    broadcast $rLow, $rLow
6    getid $rId, $rLow
7    broadcast $sp, $sp
8    $fp = $sp - (STACKLET_SIZE * TCU_ID)
9    $fp.child = NULL
10 spawn_start:
11   chkid $rId, $grHigh
12
13     bcast live regs
14     $fp = $sp - (STACKLET_SIZE * TCU_ID)
15     $sp = $fp - STUB_SIZE - frame_size
16     BlockAssembly($)
17
18   move $rId, 1
19   ps   $rId, $grLow
20   jump spawn_start
21   join
```

Figure 15: Compiling a flat spawn with cactus stack support

Finally, some stacklet set-up code needs to be injected at the onset of each parallel section to initialize the stack pointer (`$sp`) and the stacklet pointer (`$fp`) of each TCU. We revisit Figure 10 adding the stacklet code in Figure 15. Lines 7-9, inserted by the post-pass, initialize to `NULL` the child pointer in the stacklet stub of each TCU. We could have initialized these pointers before switching to parallel mode, but then would have had to perform this sequentially for each of the $\#TCU$ stacklets. Note that these child pointers must be initialized before the task code starts on Line 11 because we want to execute it once per TCU per parallel section, not once per task. Executing it once per task would defeat the purpose of keeping these pointers because it would drop the stacklets allocated to a TCU while it was working on a previous task. Lines 14-15, inserted by the core-pass, show pseudo-assembly that initializes the stacklet pointer (`$fp`) and the

[4]This separation is arbitrary and a result of what was simpler to implement.

stack pointer at the beginning of each task. This could probably be done once per TCU instead of once per task, but the additional overhead is minimal (3 cycles if the stacklet size is a power of 2), and the post-pass, which has access to the per-TCU initialization code (Lines 4-9), does not currently know the size of the activation frame, needed on Line 15. While the post-pass could recover that information, the effort did not seem worthwhile.

So far we have discussed the implementation of the cactus stack in the absence of nested parallelism, but in fact the stacklet design supports nested parallelism without any further modifications. The key functionality required for nested parallelism is having multiple child frames for a single parent frame, which stacklets support by explicitly maintaining the arguments pointer, as described.

6.3 Function Insertion

As mentioned in Section 3.3, in the current prototype of XMT, *all* the code that may be executed by the TCUs must be laid out between the `spawn` and the `join` instructions, including the code of functions that may be called. A pass called *function insertion* builds a call graph at link-time (in the post-pass), and for each outer spawn block, inserts before the join instruction the code of the functions that may be called. This is different to function inlining because the code of the function is simply placed within the spawn-join that may call it, and not substituted at every call-site.

This pass is simple enough, but because a function may be called from different spawn statements within a code, the inserted clones of the functions must be updated to have unique assembly labels. Remember that function pointers are not supported in XMTC, otherwise complex pointer analysis would be needed to build the call-graph. However, as we will see, nested spawns get compiled to code that uses function pointers in a limited way. To accommodate that, we recursively include in the call-graph functions whose address is taken within the parallel code, without performing any further pointer analysis.

6.4 Dead Function Elimination

So far, we have created two clones for each function. We then inserted function clones in the assembly between the spawn and join instructions for each outer spawn statement. This creation of many clones for each function blows up the size of the code. To remedy that, we eliminate all the function clones that cannot possibly be called by an application, after the function insertion pass. Among others, this includes all parallel clones of functions because only the inserted clones (see Section 6.3) are actually called. This helps to reduce significantly the size of the produced binary and `.sim` files, and to reduce the compilation time, as the post-pass is the slowest component, even though it performs the simplest analyses.

6.5 Outlining Optimizations

The outlining pass isolates outer spawn statements so that the core-pass (GCC) does not perform illegal optimizations, as discussed in Section 3.1. Outlining introduces inefficiencies by creating a function call where there was none. One way to take advantage of this additional function call is to pass global variables as arguments as well, in a controlled way. At first, this sounds counter intuitive: why pass as arguments variables that are global and therefore directly accessible by the tasks? The reason is that, according to the XMT calling convention (inherited from MIPS), the first four arguments to a function are passed in registers. We can take advantage of this convention by passing as arguments global variables that are read-only in the task code, as long as we do not exceed four arguments. These global values will be passed through registers and then broadcast to the tasks, hence avoiding the creation of memory hot spots by having each task access them separately.

Currently, this optimization is not path-sensitive, as it will not try to select variables that may be accessed more frequently (e.g., ones accessed in loops and ones not accessed in conditionally executed code). This would be beneficial when the candidate global variables exceed the four arguments passed through registers. A different optimization would be to explicitly load these values in registers in the outlined function, so that they can be broadcast. Such an optimization is harder to implement, however, as it needs both high level

information about the code and low level register allocation information available in the back-end of GCC. An easier alternative would be to load such variables in the read-only cache at the cluster level. The global variable would then be accessed once by each cluster, which is not as good as just once by the Master TCU, but it does not increase register pressure. This is not currently implemented.

Another potential optimization has to do with reducing the number of arguments the outlined function needs, in order to make space for passing and broadcasting global values, as per the previous optimization. Sometimes, the low and high expressions determining the number of tasks of a spawn statement can be complex, involving multiple variables, as shown in Figure 16(a). In that case, hoisting those expressions, as shown in Figure 16(b), can reduce the number of arguments significantly. Of course, if performed naively, hoisting can increase the number of arguments needed by the outlined function by two. A simple solution is to run outlining once with hoisting and once without, and to select the option that requires fewer arguments. This is not currently implemented, and therefore, the hoisting pass is not enabled by default.

(a) original code

```
spawn (a+b*c/d, e+f*g/h){ // 8 args
    task_code($)
}
```

(b) after hoisting

```
low = a+b*c/d;
high = e+f*g/h;
spawn (low, high){ // 2 args
    task_code($)
}
```

Figure 16: Hoisting Low & High Expressions

Other ways of reducing the overhead of outlining include *(1)* better analysis for reducing the number of arguments (e.g., finding the smallest number of variables or subexpressions to describe all the incoming and outgoing values of the outlined function), and *(2)* inlining back these functions at link-time. However, inlining at that late stage, after register allocation has been performed, can be challenging, and we have not explored that possibility.

6.6 Compiling a nested spawn statement

So far, we have described support for calling functions from parallel code and the necessary cactus stack support. In this section, we will add the necessary components to support nested spawn statements. Algorithm 7 shows pseudocode of a parallel clone of a function with a parallel loop (e.g., spawn statement). To allow potentially executing some of the tasks of that parallel loop on other workers, we perform a different type of outlining for nested spawns. This time, we only include the code of the task, excluding the spawn statement, since the goal is to be able to run one or more tasks independently. Algorithm 8 shows the resulting pseudocode which includes two functions: the original function `PAR_FOO` that has been modified in several ways, and the outlined function that executes the task code *grain* times.

Algorithm 7 Outlining Inner Spawn: original code

```
1: procedure PAR_FOO(params_foo)
2:     locals_foo
3:     SomeCode
4:     for all $ ∈ [low, high] do
5:         CODE[params_foo, locals_foo, $]
6:     end for
7:     RestOfCode
8: end procedure
```

Let us focus first on the outlined function (Lines 17-23). The dollar sign represents the (register-allocated) task identifier. The scheduler sets its value before invoking the outlined function. The *grain* argument allows executing multiple tasks with one invocation, a very useful feature for fine-grained codes. The *frame*

Algorithm 8 Outlining Inner Spawn: resulting code

```
1:  procedure PAR_FOO(params_foo)
2:      frame = (state_regs, rop, nChildren, tid, params_foo, locals_foo)
3:      frame ← params_foo
4:      SomeCode[frame]
5:      frame.nChildren ← high − low + 1                    ▷ Start of Inserted Code
6:      frame.tid ← $
7:      frame.state_regs ← (sp, fp, argp, rop)
8:      taskDescriptor ← (low, nChildren, frame, outlinedNestedSpawn, grain)
9:      nExec ← schedulerExecute(taskDescriptor)
10:     atomic{frame.nChildren ← frame.nChildren − nExec}
11:     if nChildren ≠ 0 then
12:         Jump to Scheduler Code
13:     end if
14:     ClobberRegisters                                    ▷ End of Inserted Code
15:     RestOfCode[frame]
16: end procedure
17: procedure OUTLINEDNESTEDSPAWN(grain, frame)
18:     end ← $ + grain                  ▷ $ ← td.low by the scheduler before calling this function
19:     while $ < end do
20:         CODE[frame.params_foo, frame.locals_foo, $]
21:         $ ← $ + 1
22:     end while
23: end procedure
```

argument gives access to `PAR_FOO`'s local variables and parameters. The task code (Line 20) is updated to access those variables through the frame pointer. The *frame* is defined separately for each function containing a `spawn`, as a structure (i.e., a C `struct`) that contains all the local variables of `PAR_FOO`, its parameters, and some additional fields for bookkeeping, which we will cover later.

The code of `PAR_FOO` is also updated to access its variables through the frame structure (Lines 4 and 15). The overhead of doing so is smaller than what one would originally assume because `PAR_FOO` will access the fields of the *frame* structure directly and not through a pointer. This means that the compiler can access any of the fields of *frame* using its stack pointer, as *frame* is now a local variable in `PAR_FOO`. Moreover, the compiler can register-allocate fields of *frame* like it would the variables of the original instance of `PAR_FOO`. Hence, the main overhead of this transformation is copying `PAR_FOO`'s parameters into the *frame* structure (Line 3). Copying is necessary because, as we mentioned, the arguments of a function may not lie directly above its activation frame, due to the cactus stack implementation.

```
typedef struct {
    void (*rop)(void);   // Rest Of Parent pointer
    int sp;              // stack pointer
    int fp;              // frame pointer
    int argp;            // args pointer
    int childNR;         // number of pending tasks
    int tid;             // task ID of parent task
} genericFrame;
```

Figure 17: Common Preamble of all frame structures.

Besides local variables and arguments, the frame structures of all functions that have a parallel loop also

have a common preamble (Figure 17). This preamble is used by the scheduler for synchronization and for allowing the worker executing the last pending task of the parallel loop to resume the parent task. In that case, the preamble is needed only if the worker resuming the parent task is not the one that initiated the spawn statement. The scheduler code can type-cast any frame to a `genericFrame` in order to access the fields of the preamble because all frames have the same preamble. Note that the first argument is a function pointer that takes no arguments. It is used to jump to the next instruction after the parallel loop (i.e., spawn statement), which is Line 14 in Algorithm 8. Line 14 clobbers all but the state registers (`$sp`, `$fp`, `$argp`, `$`) so that GCC will restore any register-allocated values from the stack. This seems unnecessary as the address of the *frame* is passed to the scheduler on Line 9, and therefore, any register-allocated fields of *frame* need to be written out to memory before that call and restored afterwards. However, GCC may decide to restore these registers soon after returning from the scheduler code, before Line 14. In that case, a remote worker jumping to that line will start executing code which incorrectly assumes that some of the fields of *frame* reside in registers. Clobbering the registers at that point ensures that GCC will not try to restore fields of *frame* into registers too soon.

```
typedef struct {
    int tId;          // TID of first thread
    int tNr;          // Number of tasks
    int grain;        // Grainsize
    void *frame;      // Pointer to parent frame
    /** The address of the code to execute.
     * The 1st argument is the grainsize.
     * The 2nd argument is the pointer to the frame
     * of the parent to access its variables. */
    int (*code_addr)(int grain, void* frame);
} TaskDescriptor;
```

Figure 18: The task descriptor structure

Now, let us focus on the code that replaced the nested spawn statement (Lines 5-13 of Algorithm 8). First, the number of child tasks is initialized in the *frame* (Line 5), and the task ID of the current task is saved (Line 6). Then, the rest of the frame preamble is filled (Line 7). Remember that the *rop* is the address of Line 14. A task descriptor structure is then initialized on Line 8. A task descriptor is akin to a *closure*: it contains all the information needed to execute some code. Here, the code is a set of tasks. Figure 18 shows the fields of the task descriptor of a parallel loop. The ID of the first task to execute along with the total number of tasks define the range of tasks included in a task descriptor. The pointer to the *frame* allows the task code to access its parent's local variables, and a function pointer to the task code allows to execute tasks. Lastly, the *grain*, which is computed by the static coarsening pass or is explicitly provided by the programmer, allows the task code to execute multiple tasks with a single invocation.

After this initialization of the parent's frame preamble and of the task descriptor, the scheduler is called on Line 9 (c.f., [16]) to schedule and execute the task descriptor. The scheduler returns the number of tasks that were executed by the present worker (*nExec*). If some tasks were placed on the deque, *nExec* will be smaller than *nChildren*, the number of tasks originally in the task descriptor. Since other workers may be working on a subset of those tasks after stealing them, they will have access to *frame.nChildren*, which they will also decrement, as they complete tasks. For that reason, the operation on Line 10 is performed atomically. In Section 6.7, we will present the code that decrements *nChildren* in the case of stolen tasks executing remotely, as well as how the thief resumes the parent task if needed (Algorithm 9). Finally, the parent task checks if all its tasks have completed (Line 11) and resumes if that is the case (Line 14) or jumps to scheduler code responsible for feeding the worker with more work. In work stealing, the main family of schedulers implemented in XMTC, that is the code for consuming work from the local deque or for stealing code from a victim worker. Note that the worker does not *call* the scheduler code, it simply jumps to it, relinquishing its stack to the worker that completes the last of the child-tasks and becomes responsible for

resuming the parent task.

6.7 Outer Spawn Compilation for Nesting

Now, we want to combine XMT's hardware scheduling of outer spawn statements with software scheduling for nested parallelism. The first step is to convert the spawn statement, as done in Figure 19. Depending on the task ID ($), the worker will either execute it, or turn to the software scheduler for work if the task ID is not a valid outer task ($ > *high* on Line 2). In the current implementation, incrementing *high* within the spawn block is not supported because our goal is to support structured nested parallelism. Therefore, the legacy single-spawn (`sspawn`) is not supported in conjunction with nested parallelism. Moreover, since *high* cannot be increased, TCUs will keep requesting work from the scheduler once outer tasks are consumed, hence the infinite loop on Line 4.

(a) Original Code

```
spawn (low, high) {
    CODE[$]
}
```

(b) Converted Code

```
1  spawn (low, high) {
2      if ($>high) {   // software sched.
3          Allocate Stacklet
4          while(true) {
5              Get Work from scheduler
6              Execute it
7              If(last) resume parent
8          }
9      } else {        // hardware sched.
10         CODE[$]
11     }
12 }
```

Figure 19: Outer Spawn Conversion

Before we elaborate on the stacklet allocation (Line 3) and on the body of the scheduling loop (Lines 5-7), we address the issue of detecting global termination of a parallel section in the presence of nested parallelism. We use the same XMT hardware support as we did for a flat spawn statement in Figure 10, but we use it differently, as shown in Figure 20. We use three global registers instead of two. The use of `$grLow` is the same as before; it stores the next available task ID for outer tasks. However, we do not use `$grHigh` in a `chkid` instruction to detect the global termination of the parallel section, as nested tasks may be pending even if the last outer task has completed. Instead, we use two global registers: `$grProduced` (Line 2), which is increased each time a task descriptor is added onto the scheduler's work-pool, and `$grConsumed` (Line 1), which is increased each time a task descriptor is removed from the scheduler's work-pool. The difference of `$grProduced-$grConsumed` gives the number of available task descriptors in the work-pool. Thanks to XMT's hardware prefix-sum support, this information can be maintained at very low cost and in a scalable way. We use `chkid $grConsumed, $grProcuced` to block workers when the work-pool is empty[5]. The reason for using two global registers is that the hardware prefix-sum operation does not accept a decrement at the moment. If it did, we could simply keep the total number of task descriptors available in the work-pool. Such support would be needed in a production quality XMT to avoid overflow, but for the current prototyping stage, it has not been a limitation in practice. Note that `$grConsumed` is initialized to 1 instead of 0 (Line 1) because `chkid` blocks only when its first argument is strictly greater than its second argument. When they are equal, execution proceeds with the next instruction.

Using XMT's `chkid` instruction allows to block TCUs when there is no work in the work-pool. In work stealing, where the work-pool is distributed and workers have to randomly probe the deques of other workers, the `chkid` mechanism prevents wasted probing, which can potentially be harmful for performance

[5]In reality we first copy the value of `$grConsumed` into a local register using a prefix-sum with a zero increment, because `chkid` takes a local register and a fixed global register as arguments.

```
1    mvtg  $grConsumed, 1
2    mvtg  $grProduced, 0
3    getid $rTmp, $rLow
4    mvtg  $grLow, $rTmp
5    spawn
6    broadcast $rLow, $rLow
7    getid $rId, $rLow
8    broadcast $sp, $sp
9    $fp = $sp - (STACKLET_SIZE * TCU_ID)
10   $fp.child = NULL
11 spawn_start:
12   bcast live regs
13   $fp = $sp - (STACKLET_SIZE * TCU_ID)
14   $sp = $fp - STUB_SIZE - frame_size
15   if ($ > high) {
16   } else {
17     BlockAssembly($)
18   }
19 join_label:
20   call
21   move $rId, 1
22   ps   $rId, $grLow
23   jump spawn_start
24   join
```

Figure 20: Compiling an outer spawn to support nesting

by issuing useless memory requests. Moreover, the `chkid` mechanism effectively notifies idle workers when a task descriptor is added so that they can resume work stealing. Work stealing implementations on platforms that do not have such hardware support employ methods such as exponential back-off after a certain number of unsuccessful deque probes to avoid adversely affecting the performance of busy workers.

Algorithm 9 Scheduling loop for work stealing

```
1:  while true do
2:      success ← stealWork(&taskDescriptor)
3:      if success = false then continue with next iteration
4:      end if
5:      nExec ← schedulerExecute(taskDescriptor)
6:      gFrame ← (_xmt_generic_frame) taskDescriptor.frame
7:      atomic{gFrame.nChildren ← gFrame.nChildren − nExec}
8:      if gFrame.nChildren = 0 then                          ▷ Resume Parent Task
9:          $sp ← gFrame.sp                                   ▷ Restore Stack Pointer
10:         $fp ← gFrame.fp                                   ▷ Restore Frame Pointer
11:         $argp ← gFrame.argp                               ▷ Restore Arguments Pointer
12:         $ ← gFrame.tId                                    ▷ Restore Task ID
13:         jump to gFrame.rop                                ▷ Jump to Rest of Parent Task
14:     else
15:         Keep executing work of own deque
16:     end if
17: end while
```

Algorithm 9 expands Lines 5-7 of Figure 19(b) for work stealing schedulers. These include Cilk's scheduler,

TBB's simple-partitioner, auto-partitioner, and affinity-partitioner (not implemented in XMTC), and the lazy work stealing variants described in [16]. First, the worker attempts to steal a task descriptor until it succeeds (Lines 2-3). Then, the scheduler executes the descriptor, which involves the possibility of splitting it and pushing part of it on the local deque (Line 5). Upon return, the worker atomically decrements the number of children of the parent task through the frame pointer stored in the descriptor (Lines 6-7), and if the worker just completed the last task of the parent task, it resumes it (Lines 8-13). Otherwise, the worker consumes the local deque (Line 15) that may have been filled by the call on Line 5 if the task descriptor was split.

Algorithm 10 Stacklet Allocation for Work-Stealing Loop

```
1: atomic{$fp ← Global_SP; Global_SP ← Global_SP - STACKLET_SIZE;}   ▷ Implemented with psm
2: $fp.end_address ← $fp - STACKLET_SIZE;
3: $fp.child_stacklet ← NULL
4: $fp.parent ← NULL
5: $fp.saved_sp ← NULL
6: $fp.saved_argp ← NULL
7: $fp.return_address ← NULL
8: $sp ← $fp - (STUB_SIZE + frame_size)
```

Finally, Algorithm 10 expands the stacklet allocation of Line 3 of Figure 19(b).

7 Conclusion

In this technical report, we presented a series of lessons for compiler developers using a compiler for a sequential language to compile a parallel language, and we gave an in-depth description of the XMTC compiler internals, including how to outline tasks, how to take advantage of XMT's hardware primitives for scheduling nested parallelism, how to implement a cactus stack using stacklets on XMT, and how to perform different types of prefetching on manycores with very small prefetch-buffers and without private coherent caches.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proc. Conference on Programming Language Design and Implementation*, 2005.

[3] George Caragea, Alexandre Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for many-cores. In *Proc. Intl. Symposium on Parallel and Distributed Computing*, 2010.

[4] George C. Caragea. *Optimizing for a Many-Core Architecture without Compromising Ease-of-Programming*. PhD thesis, University of Maryland, College Park, 2011.

[5] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for fine-grained many-cores. *International Journal of Parallel Programming*, 39(5):615–638, 2011.

[6] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. In *Proc. Technology of Object-Oriented Languages*, 1998.

[7] Seth Copen Goldstein. *Lazy Threads Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California–Berkeley, Berkeley, CA, 1997.

[8] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[9] Fuat Keceli. *Power and Performance Studies of the Explicit Multi-Threading (XMT) Architecture*. PhD thesis, University of Maryland, College Park, 2011.

[10] Fuat Keceli, Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Toolchain for Programming, Simulating and Studying the XMT Many-Core Architecture. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (in conjunction with IEEE IPDPS)*, Anchorage, Alaska, USA, May 2011.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[12] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proc. ACM Symposium on Principles of Programming Languages*, 2005.

[13] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 62–73, New York, NY, USA, 1992. ACM.

[14] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[15] Diego Novillo. Tree ssa: A new optimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[16] Alexandros Tzannes. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. PhD thesis, University of Maryland at College Park, 2012.

[17] J. Wagner, A. Jahanpanah, and J.L. Träff. User-land work stealing schedulers: Towards a standard. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 811 –816, march 2008.

[18] Xingzhi Wen. *Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) Paradigm*. PhD thesis, University of Maryland, College Park, 2008.

[19] Xingzhi Wen and Uzi Vishkin. PRAM-on-chip: first commitment to silicon. In *Proceedings of the annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[20] Xingzhi Wen and Uzi Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 2008 conference on Computing frontiers - CF '08*, page 55, May 2008.