Automatic XMTC to Cilk Translation

Alexandros Tzannes

December 4, 2007

Abstract

Parallel programming is known to be notoriously difficult! Be it hardware or compiler limitations that dictate what a parallel programming language supports or the fact that different applications benefit from different types of parallelism, no programming paradigm has emerged a clear winner in the parallel arena. In this work we start building a bridge between two programming approaches, XMTC and Cilk, by providing an efficient source-to-source translation from XMTC to Cilk.

1 Introduction

For at least 30 years there has been relentless work on computer parallelism for general purpose or super-computing applications. The idea of getting multiple tasks processed in parallel is an old idea but, maybe surprisingly, to this day, there is no universally accepted model for parallel computation. So far all the models that were proposed were either deemed too abstract (e.g., PRAM [3]) or exposed too many details to the programmer, making it a very hard task to reason about correctness and performance of a parallel program (e.g. MPI [4]). All the different approaches have strong proponents but none of them dominates. For example shared-memory vs. message passing models have battled for a long time with no definite winner. Our work focuses on the shared-memory approach, not because we claim it to be universally superior, but because, in light of the arrival of multicores where shared-memory is the natural choice as it is supported in hardware, we believe that making the most of such readily available hardware is crucial.

Cilk [5] is a multi-threaded shared-memory language. It extends C by adding a small number of new keywords which makes it easy to learn. Moreover the process of changing a Cilk program into a C program is a trivial mechanical process. Cilk is build to run on systems that support POSIX threads. It is not meant to run on networks of workstations although there has been some work in that direction. The main strength of Cilk lies in its natural way of expressing divide-and-conquer algorithms which are prevalent in AI (e.g., any graph traversal).

In our opinion, the main limitation of Cilk is that the only primitive introducing parallelism, is calling and executing a function in a new thread. This effectively creates one new thread, but in many cases the available amount of parallelism is far larger. Assume for example that we would want to check if any of ten thousand elements of an array satisfies a predicate. In Cilk the best way to implement this is to follow a divide and conquer paradigm: recursively spawn off two threads, one to check the left half of the array and one to check the right half. A more natural way to code this would be to allow the programmer to express all of the available parallelism: start simultaneously ten thousand threads. This is where XMTC comes in.

In XMTC parallelism is introduced by a construct also called `spawn` but which starts any number of threads specified by the programmer. The philosophy of XMTC is that the programmer should express all the available parallelism, and it is then the responsibility of the compiler and the runtime system to run the program efficiently. This removes the burden of writing code following constraints imposed by the hardware and improves productivity. Unfortunately until this writing, XMTC code can only be run on a simulator of the XMT architecture and an XMT FPGA [12]. Moreover the simulator is very slow when run in cycle-accurate mode and the FPGA is mainly a proof of concept and has limitations for practical programming (e.g., does not currently support floating-point operations, has only (!) 64 cores, and runs at 75MHz).

This is where the XMTC to Cilk source-to-source transformation fits in. Our tool allows programmers to write a program which is more naturally expressed in XMTC in that language and run it on commodity hardware, without incurring unreasonable overheads.

2 Background

2.1 Cilk

Cilk [5] has been developed at MIT for over a decade. It introduces parallelism by allowing to call a function in a new thread (also called a future) and resuming execution immediately after the call, without having to wait for the result of the function. A way to explicitly wait on the results of all pending futures is also provided. This “top-down” model starts with one thread and each active thread can `spawn` at most one additional thread at each time-step. This is particularly well suited to divide-and-conquer algorithms where the *combine* step needs to combine two (or a small constant number) of partial results. Consider the parallel summation algorithm where we have to add n values and we use a binary tree. The initial values start at the leaves and the result will be stored in the root. The computation proceeds in rounds, one for each level of the balanced binary tree. At each level, all nodes in parallel read and add the values of their two children, and store the result in themselves. We call the Cilk model “top down” because for the parallel computation of summation (which many parallel algorithms follow) the computation recursively descends the tree from the root to the leaves, and when the recursion backtracks the values are propagated towards the root.

2.2 XMTC

XMTC [2] is developed at the University of Maryland, College Park. It introduces Single Program Multiple Data (SPMD) parallelism via the `spawn` construct which takes as arguments the number of threads to start and the code they should execute. After *all* the spawned threads have terminated, the execution continues at the statement after the `spawn` . In other words, there is an implicit synchronization point at the end of the parallel code of the spawn statement. Currently implementation (and hardware) limitations disallow the use of function calls inside the parallel code, which excludes recursion in parallel-mode, like Cilk does. The XMTC implementation of the summation algorithm starts from the leaves and proceeds in rounds moving towards the root. We call this model the "bottom-up" computation model. The disadvantage of this approach for this particular problem is that each round has to finish before the next round starts, even though some elements in the next level could already be computed because both their children have already been computed (this can be coded in XMTC but takes a bit more effort).

2.3 Example

In this section we will give a simple example of the transformation to illustrate the source-to-source transformations that are needed.

In Listing 1 we see the XMTC code of a function that takes 2 arrays as arguments and their size, and copies the first array to the second one. For each element to be copied a thread is spawned. In Listing 2 we have the naive transformation from the XMTC code. The code of the body of the spawn is closure converted with the special dollar symbol acting as a parallel induction variable replaced by an argument to the closure. We will refer to this transformation as *outlining*, the reverse operation of inlining. This is a correct but naive way to translate XMTC to Cilk because typically the XMTC threads are very fine grained and the thread overheads in Cilk are substantially higher.

Listing 1: XMTC array copy

```
int arrayCopy(int A[],int B[], int size) {

    spawn (0,size -1) {
        B[$] = A[$];
    }
    return 0;
}
```

In Listing 3 we see a less naive transformation. In this code $NPROC$ takes the value of the global Cilk environment variable $Cilk\_active\_size$ which denotes the number of available processor and is a runtime constant. So this transformation creates one clustered thread per active processor and each clustered thread

Listing 2: Cilk array copy (naive translation)

```
cilk int arrayCopy(int A[], int B[], int size) {
    int i;
    for(i=0;i<size;i++) {
        spawn outlinedCode(A,B,i);
    }
    sync;
    return 0;
}

cilk void outlinedCode(int A[], int B[], int i) {
    B[i] = A[i];
}
```

contains, in a for-loop, multiple *thin* threads (i.e., original XMTC threads).

Listing 3: Cilk array copy (clustered translation)

```
cilk int arrayCopy(int A[], int B[], int size) {
    int i;
    int NPROC = Cilk_active_size;
    int a; // number of thin threads per clustered thread
    a = ceil(size/NPROC);
    for(i=0;i<NPROC;i++) {
        spawn outlinedCode(A,B,i,a);
    }
    sync;
    return 0;
}

cilk void outlinedCode(int A[], int B[], int i, int size) {
    int L = i*a;
    int U = min{(i+1)*a, size);
    int j;
    for(j=L; j<U; j++)
        B[j] = A[j];
    return 0;
}
```

3 Implementation

The source-to-source translation is implemented as a number of passes in CIL [10] which we had to extend to allow support for XMTC. Fortunately extensions for Cilk were not necessary. We proceed to describe the modifications we made to the CIL infrastructure, then we present the outlining pass and finally discuss the clustering pass.

CIL modifications The extensions needed to support the XMTC constructs were relatively modest. First we needed a new type of statements for the *spawn* statement. Adding it to the lexer and parser was relatively easy. What was longer was extending the internal CIL data structures to allow for this new statement kind and correctly transforming from the abstract syntax-tree structure (Cabs) to the CIL data structures. The rest of the XMTC-specific constructs were added through a library include with *asm* statements.

The second and more important modification, which set back the whole project, was the default behavior of CIL to flatten variable declarations within a function, and put them all at the top of the function. While correct for C code, this transformation was incorrect for XMTC programs because variables declared inside a spawn statement had different semantics: they are thread local and each thread has its private copy. Moving such a declaration to the top of the function makes it a shared variable. Changing this behavior involved modifying CIL significantly and moving the declarations to the enclosing block instead of the enclosing function.

Another modification we had to make was to prevent CIL from compiling the resulting Cilk file using GCC, since its scripts were trying to do that automatically, and there was no obvious option to prevent that. Additionally all of the passes that came with CIL were turned off, since they were not aware of the spawn construct and many of them would have not worked.

Outlining Pass Once CIL was in extended as described, the next step was to implement the closure conversion pass. The first task is to determine which variables are used in the parallel code and which of them are defined in the serial part of the code and thus need to be passed as arguments. One exception to this rule was global variables which are accessible in the outlined function anyways and don't need to be passed as arguments. One thing we are currently not doing is checking whether these variables we are passing as arguments are updated by the parallel code or not. If they are, we should pass them "by reference". This is a limitation but not an important one since arrays which are typically updated in parallel XMTC code are passed by reference anyways.

After the arguments are determined, the outlined function is created and the spawn statement is replaced by a for-loop that spawns serially all the parallel threads and then waits for all of them to terminate. One thing to keep in mind is that in Cilk there is an implicit `sync` just before the return statement of every function, so in some cases it can be omitted.

Clustering We implemented a simple version of clustering that clusters all the threads into $NPROC$ threads. The first $NPROC - 1$ clustered threads

Listing 4: before clustering

```
spawn(lo,hi) {
    CODE($)
}
```

Listing 5: after clustering

```
int T = Cilk_active_size;
int a = ceil((hi-lo+1)/NPROC);
spawn(0,T-1) {
    int L = lo + $*a;
    int U = min(L+a,hi+1);
    int i;
    for(i=L;i<U;i++) {
        CODE(i)
    }
}
```

Figure 1: Generic Clustering Transformation

get $\lceil \frac{\#ThinThreads}{NPROC} \rceil$ thin threads and the last clustered thread gets the remaining thin threads. This creates a slight imbalance since the difference of thin threads between the last clustered thread and the rest of them can be as large as $NPROC - 1$. Alternatively we could have implemented the clustering in a way to assure that any two clustered threads have the same number of thin threads, give or take one. We chose the first because it introduces slightly less overhead and we assume the number of processors to be relatively small compared to the number of thin threads (so an $NPROC - 1$ imbalance is small). In Figure 1 we show how the transformation is performed on a generic spawn statement. Notice that since this is an XMTC to XMTC transformation it has to happen before outlining.

4 Discussion

Clustering comes at a cost. First of all the amount of parallelism is reduced since we are effectively reducing the number of threads. This can be an important drawback when the length of the threads is non uniform: if we have $NPROC$ threads and one of them is substantially longer than the rest, all but one processors will remain idle while waiting for the long thread to terminate. The advantage of having more short threads is that they can be dynamically scheduled so as to keep most or all of the processors constantly busy, which achieves *load balancing*.

For this project we will not worry about load balancing that much since we will investigate embarrassingly parallel programs that are better expressed in

XMTC. In those programs the threads typically have few control structures in the parallel mode (if statements and loops) that would make some threads run substantially longer than others.

5 Experiments

All the experiments were embarrassingly parallel programs which are better expressed in XMTC rather than in Cilk. The resulting Cilk programs (and serial versions of the programs) were run on an Intel Core 2 Duo T7200 (2.00GHz, 4MB L2 Cache, 677MHz FSB) with 1GB of RAM. The amount of RAM is not significant though since the datasets were small and when it was needed we ran the program in a loop to increase the runtime. We run two sets of experiments, the first with the default level of optimizations and the second one with the -O4 flag. The speedup was computed as the execution time of the serial program over the execution time of the Cilk program produced by our tool, when run on 2 processors. The slowdown is the percentage of the difference of the runtime between the Cilk program on one core and the serial program, over the runtime of the serial program ($Slowdown = 100 \cdot \frac{T_{Cilk1core} - T_{serial}}{T_{serial}}$). All the execution times are reported in seconds. As we will see the runtime of the clustered version of our programs ran in almost the same time as the serial processor, when ran on one processor. This happened because our clustering algorithm effectively serializes all the execution when run on a single processor (i.e., only one thread is spawned!).

Array Copy The first program was array copying. The XMTC program spawns one thread per array element and copies that element. We run the experiment for two sizes, 10,000 elements and 100,000 elements. Both were wrapped around a loop that executed the array copying 100,000 times.

For the small data set we run our tool both with the clustering option turned off (denoted by *(nc)*) and on. As expected, because the number of threads is so large and their computation so trivial, without clustering the overhead of spawning threads dominates. In fact on two cores the version without clustering ran 55 times slower than with clustering. Moreover the execution times for one and two cores for the unclustered versions were not much different. When the execution on two cores was running, one core was active spawning threads and the other was executing them at a much faster rate, staying idle most of the time. This clearly happens because all the threads are spawned in a serial loop. If we compared with a divide and conquer Cilk algorithm we would find that both cores would be active (even though the performance would still be very poor).

Also, when the -O4 level of optimizations was used, the program ran faster on one core than on two. This is not surprising because of how the internal scheduling of Cilk works: the initial thread spawns a great number of small threads and the second processor tries to *steal* work from the bottom of the stack of the initial thread. This thread stealing introduces overheads that in the case of -O4 optimizations does not outweigh the benefit of running some of the work on the second processor.

One other important thing to notice is that with clustering, the Cilk program when ran on a single core ran less than 11% slower than the serial program compiled with GCC. When we ran the same Cilk program on two cores we got a speedup of 1.62 for the small dataset and 1.77 for the larger one.

When using optimization level -O4, we got a very small speedup for the small dataset and a decent one for the larger one. One important point is that the serial slowdown (Cilk code on one core vs. serial code) was less than 12%.

	Array Copy		Array Copy (-O4)	
	10K ($\times$100K)	100K ($\times$100K)	10K ($\times$100K)	100K ($\times$100K)
Serial	4.50 s	44.80 s	0.93 s	9.36 s
1 Core	4.80 s	49.40 s	1.04 s	10.3 s
2 Cores	2.77 s	25.27 s	0.83 s	5.66 s
Speedup	1.62	1.77	1.12	1.65
Slowdown(%)	6.67%	10.27%	11.94%	10.01%
1 Core (nc)	172.81 s	–	41.68 s	–
2 Cores (nc)	152.70 s	–	43.42 s	–

Matrix Multiplication This is a simple implementation of Matrix Multiplication, where $A[N][M]$ is multiplied to $B[M][N]$ to give the result $R[N][N]$. There is no explicit or compiler-inserted blocking in the code, which is a techniques commonly used for this problem to improve cache locality. The serial code has three nested for loops. The XMTC code spawns N threads, one for each row of the result array and each thread performs $N * M$ multiplications. Nested spawns are not supported at this time. If they were, the natural way to write this program would be to spawn N^2 threads, one for each element of the array R. In that case each thread would perform M multiplications.

We run matrix multiplication on two datasets, one with $(N, M) = (1024, 1024)$ and one with $(N, M) = (2048, 2048)$. Since we are multiplying integers we do not bother initializing the arrays because the number of cycles for an integer multiplication does not depend on the values of the operands.

There are a couple of interesting observations for this problem. First, clustering does not affect the performance. This happens because each thread has $\mathcal{O}(n^2)$ work, in other words it is relatively long so the overhead for spawning threads is but a small fraction of the computation time. Second, when we used the -O4 level of optimizations we got a speedup of more than 2 for 2 cores and a negative slowdown. After running the experiment several times, we concluded this was within the error margin and do not hold any statistical confidence. The only conclusion can be that the clustered and unclustered versions have similar and near optimal performance.

	Matrix Multiplication		Matrix Multiplication (-O4)	
	1024×1024	2048×2048	1024×1024	2048×1024
Serial	7.70 s	62.19 s	5.55 s	44.66 s
1 Core	8.46 s	69.15 s	5.64 s	44.61 s
2 Cores	4.30 s	34.74 s	2.88 s	22.24 s
Speedup	1.79	1.79	1.92	2.01
Slowdown(%)	9.86%	11.19%	1.71%	-0.11%
1 Core (nc)	8.59 s	69.52 s	5.56 s	44.05 s
2 Cores (nc)	4.37 s	34.90 s	2.81 s	22.17 s
Speedup (nc)	1.76	1.78	1.97	2.01
Slowdown% (nc)	11.53%	11.78%	0.16%	-1.37%

Sparse Matrix Multiplication (MatVec In this program the input is a sparse matrix and a vector and the result is a vector of their multiplication. The sparse matrix is stored in a 1-D array (only the non-zero values). There are also two auxiliary arrays. One has for each row of the sparse matrix the place in the 1-D array which holds the first element of that row. A second auxiliary array (which has the same size as the 1-D array of values), stores the column number in the matrix that the corresponding element in the 1-D array holds. The XMTC program spawns a thread per row of the sparse matrix. For this problem we only have one dataset of 30,000 elements that we wrap in a loop of 10,000 and 100,000 iterations. Unlike for the matrix multiplication, the data is initialized in this problem (which makes the compilation take a long time because it is initialized in the code).

Running the same dataset a different number of iterations didn't give us any surprise. The one with 10 more iterations take 10 times more time in all cases. The things to notice here is that again clustering makes a very big difference. The work performed by each thin thread is of the order of n (the size of the vector) but multiplied by a constant which on average is quite smaller than 1. In other words threads are too short and clustering is very beneficial. Also with the -O4 level of optimizations the slowdown went up and the speedup went down (even though the absolute performance increased substantially).

	MatVec		MatVec (-O4)	
	30K(×10K)	30K(×100K)	30K(×10K)	30K(×100K)
Serial	11.23 s	112.22 s	5.19 s	51.97 s
1 Core	11.42 s	113.82 s	6.16 s	60.65 s
2 Cores	5.78 s	58.21 s	3.16 s	30.77 s
Speedup	1.94	1.93	1.64	1.69
Slowdown(%)	1.72%	1.42%	18.70%	16.71%
1 Core (nc)	60.85 s	610.60 s	17.26 s	175.65 s
2 Cores (nc)	61.42 s	614.09 s	18.60 s	186.10 s
Speedup (nc)	0.18	0.18	0.28	0.28
Slowdown% (nc)	441.83%	444.09%	232.72%	238.00%

discussion Our experiments show that for embarrassingly parallel programs, translating them automatically from XMTC to Cilk can be done very efficiently so as to actually get significant speedups relative to the serial programs. Moreover this is done in a modular way, such that the same executable can be run on one, two, or more processors and take advantage of them (in all of our experiments the results we got for 1 Core and 2 Cores came from the same executable).

6 Related Work

6.1 Source-to-source Tools

There are multiple source-to-source tools performing different types of tasks, from program improvement [8], to atomicity for Java [6], to database query translation [7], and many more. CIL [10] is a C intermediate language that has a number of optimization passes that manipulate the program. At the end the program is written back to C before it is processed by a compiler. There are also a few tools that translate code from one programming language to another [11, 1]. Our tool is unique in that it is the first XMTC to Cilk tool. Its purpose is to demonstrate the value of adding parallel loops to a language that only supports parallel function calls.

6.2 Thread Clustering

The only previous work we found on thread clustering was in the thesis of Dorit Naishlos [9] on her work on the first XMTC compiler. The level of detail is more advanced there but clustering is used as a compiler optimization for compiling XMTC code for the XMT architecture. Clustering for our XMTC to Cilk tool follows the same principles but the details are different: the number of processors is small and not known at compile time, and the overhead of starting a thread is significantly larger.

We found some recent work on thread scheduling on multiprocessors. They also call it thread clustering but their goal is to assign a thread to a processor so that it will find as much of the data it needs locally. This completely orthogonal to what we do since we assume all of shared memory has the same latency.

In another work the authors advertise doing thread coarsening [13] which sounds related to what we do, but in their paper they get some threads as input and to satisfy a requirement of their computation model, they break them in smaller threads at compile time. As with any static analysis, they are too conservative, resulting in a large number of smaller threads, so they describe optimizations for better analysis to break the threads into larger chunks. While their motivation is the same as ours, namely to reduce thread overheads due to too short threads, the two approaches are not related.

7 Future work

Implementation Because much time was spent changing CIL so it will support XMTC, many features were not implemented. First and foremost, nested spawns in XMTC were not supported. This should be relatively easy to support since Cilk supports nested parallelism. All that needs to be done is to extend the outlining CIL pass to work in the case of syntactically nested spawns. The second limitation of our implementation is that values cannot be passed by reference to the outlined functions we create. This means that scalar shared variables that reside on the stack (global variables do not fall in this category) cannot be updated in the parallel code. Then we would also need to support the rest of the XMTC statements (`ps`, `psm`, and `sspawn`). Doing that will involve using locks in Cilk and would be an interesting challenge. Note however that Cilk was designed to work without locks; some guarantees given about the quality of the scheduling Cilk does, which is one of the major selling points of Cilk, become void if locks are used in the code.

Experiments We should run similar experiments on more programs, some including floating point operations, and some involving more irregular computations, such as Breadth-First search. Moreover we should run these same experiments on the XMT FPGA and the simulator to project what the speed of a full blown XMT machine would be. This would not serve as evidence that our implementation is good, but as evidence that the XMT architecture is well suited for such problems. We would also like to take some of the programs provided with Cilk, implement them in XMTC, use our tool to produce a Cilk program and compare the two Cilk programs. We didn't do that because all of the Cilk example programs we found were heavily hand-optimized (blocking, unrolling, etc) and there was not enough time to understand them and write an XMTC hand-optimized counterpart.

8 Conclusions

We implemented a source-to-source translation of XMTC code to Cilk code which took an XMTC program with fine-grained parallelism and created a Cilk program with more coarse-grained threads. The execution times of the Cilk program didn't fall behind the serial by much on one processor and achieved significant speedups on two processors. The results provide evidence that the Cilk language has a powerful lightweight runtime system, and that the XMTC language, which is better suited for some types of programs, can harness the power of Cilk without incurring unreasonable overheads. An other way to look at these results is that fine grained parallelism can be supported in Cilk by adding a parallel loop such as the `spawn` statement of XMTC, and by implementing a thread clustering optimization. In conclusion our work acts as a strong proof that Cilk could support a powerful SPMD construct such as the `spawn` construct of XMTC, providing near optimal performance, without any programmer assistance.

References

[1] Albrecht, P. F., Garrison, P. E., Graham, S. L., Hyerle, R. H., Ip, P., and Krieg-Brückner, B. Source-to-source translation: Ada to pascal and pascal to ada. In *SIGPLAN '80: Proceeding of the ACM-SIGPLAN symposium on Ada programming language* (New York, NY, USA, 1980), ACM Press, pp. 183–193.

[2] Balkan, A. O., and Vishkin, U. Programmer's manual for XMTC language, XMTC compiler and XMT simulator. Tech. Rep. UMIACS-TR-2005-45, UMIACS, 2006.

[3] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming* (1993), pp. 1–12.

[4] Dongarra, J. J., Otto, S. W., Snir, M., and Walker, D. A message passing standard for mpp and workstations. *Commun. ACM 39*, 7 (1996), 84–90.

[5] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. 212–223.

[6] Hindman, B., and Grossman, D. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness* (New York, NY, USA, 2006), ACM Press, pp. 82–91.

[7] Howells, D. I., Fiddian, N. J., and Gray, W. A. A source-to-source meta-translation system for relational query languages. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1987), Morgan Kaufmann Publishers Inc., pp. 227–234.

[8] Loveman, D. B. Program improvement by source-to-source transformation. *J. ACM 24*, 1 (1977), 121–145.

[9] Naishlos, D. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. Master's thesis, University of Maryland, College Park, Computer Science Department, 2000.

[10] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction* (London, UK, 2002), Springer-Verlag, pp. 213–228.

[11] Springen, N. L. A source-to-source translator from pascal to c. Tech. rep., Austin, TX, USA, 1982.

[12] Wen, X., and Vishkin, U. PRAM-On-Chip: 1st commitment to silicon. In *Proc. 19th Symp. on Parallel Algorithms and Architectures (SPAA)* (2007).

[13] Zoppetti, G. M., Agrawal, G., Pollock, L., Amaral, J. N., Tang, X., and Gao, G. Automatic compiler techniques for thread coarsening for multithreaded architectures. In *ICS '00: Proceedings of the 14th international conference on Supercomputing* (New York, NY, USA, 2000), ACM Press, pp. 306–315.