

An implementation of the SANE Virtual Processor using POSIX threads

M. W. van Tol, C. R. Jesshope, M. Lankamp, S. Polstra

University of Amsterdam, Netherlands

Abstract

The SANE Virtual Processor is an abstract concurrent programming model that is both deadlock free and supports efficient implementation. It is captured by the μ TC programming language. The work presented in this paper covers a portable implementation of this model as a C++ library on top of POSIX Threads. Programs in μ TC can be translated to the standard C++ syntax and linked with this library to run on conventional systems. The motivation for this work was to provide an early implementation on conventional processors as well as supporting work from programming FPGA chips to Grids

1 Introduction

The SANE Virtual Processor (SVP) is a programming and machine model [1] developed by the Computer System Architecture group at the University of Amsterdam in conjunction with the European AETHER project [5], based on the Microthread model [2]. The SVP is an abstract concurrent processor with a range of concurrency controls to manage schedulable entities named microthreads. μ TC [3] is an intermediate language based on C extended with these concurrency controls to program the SVP.

This paper will cover the work that has been done on implementing SVP in a C++ library using POSIX threads (in short; *pthreads*), enabling the compilation of μ TC code to a program that can run on any conventional systems that has proper *pthread* support. It is organized in the following manner: In Section 2 we will discuss the background on the SVP model and the μ TC language, followed by Section 3 discussing the motivation behind

Email address: `mwwantol,jesshope,mlankamp,spolstra@science.uva.nl` (M. W. van Tol, C. R. Jesshope, M. Lankamp, S. Polstra).

this work and what it can be used for. Section 4 goes into details of the implementation of the library and the tool chain, and in Section 5 we present the results of experiments on our implementation. In Section 6 we give an overview of the future work and conclusions.

2 Background

2.1 SANE Virtual Processor

The SANE Virtual Processor, or SVP [1], is a concurrency model to program the SANE '*Self Adaptive Networked Entity*', a concept from the European AETHER project [5]. The SVP is a more generalized form of the Microthread model, and related to the work on the Microthread architecture [ref]. Because of this relation it is close to hardware, capturing the locality of communication and synchronization, but still it has a high level of abstraction. This makes its implementation independent, as it can be implemented directly in hardware, as part of an Operating System, or, as shown in this paper, as a library on top of conventional systems.

The model is composable and works recursively across all granularities of concurrency, from high level task parallelism to low level instruction level parallelism between microthreads. All concurrency at every level is explicitly captured by the model, unlike the scheduling, which is induced by synchronizations, and done dynamically depending on the underlying implementation. The model assumes two types of memory, local synchronizing memory for data-flow like execution and inter-thread synchronization, and a globally shared asynchronous memory.

2.1.1 Microthreads

All code in SVP is executed in microthreads, which are the smallest schedulable fragments of code in the model, and the level of instruction level parallelism in microthread architectures. These threads can be of any granularity; from a few instructions to an entire sequential program, but all actions within a thread are strictly sequential.

2.1.2 Families of threads

Threads are grouped in families, which is the granularity of a unit of work, and the unit in which threads are managed from the SVP. A thread can

create a new family of threads, introducing hierarchy and recursion into the model. All threads within a family are running the same code, but each has its own sequenced index value within its family. Local communication is done through writing and reading shared variables from one thread to the next thread in sequence within a family only. The first thread in sequence will receive the shared variables from the parent thread that created the family, and the last thread in sequence will return the shareds to the parent. This restriction prevents deadlock on communication between threads.

Bulk synchronization and communication is done at the family level, as a thread that created a family of threads can synchronize on the completion of that family. Only when this synchronization has completed, the asynchronous memory modified by this family can be safely read and used.

2.1.3 Concurrency controls

Concurrency is managed at the family of threads level, as this is the unit in which threads are *created* and *synchronized*. These two operations are implemented by the *create* and *sync* control. *Create* creates a parameterized family of threads, and *sync* blocks until the target family has completed or aborted execution. There are three other controls to influence the execution of a family of threads; *kill*, *break*, and *squeeze*. *Kill* aborts the execution of a target family, whereas *break* aborts the execution of the family the thread executing the *break* is part of. *Squeeze* is a preemption control to interrupt the execution of a family, but doing this in a graceful way. A *squeezed* family will stop creating new threads, and all currently running threads will run to completion.

As the model works recursively, so do these concurrency controls. When a family is *killed*, all subordinate child families will be *killed* recursively. When a thread *breaks* its family's execution, all child families are aborted similarly to a *kill*. The special case is *squeeze* where it can be defined if the *squeeze* preemption should be propagated to child families or not; this gives the programmer the control to make sure a family can be cleanly interrupted with *squeeze*.

2.2 μ TC language

2.2.1 Syntax

The μ TC language [3] is an extension to the C99 [ref] standard, which captures the SVP model as presented in the previous subsection. It has additional types to define e.g. shared variables, family identifiers, and thread index. Keywords corresponding with the concurrency controls have been added, as well

as a special thread declaration block of code, similar to how a function in C is defined.

2.2.2 Tool chain

Several tools are developed or are in development around the μ TC language. This tool chain includes source to source compilers compiling from the Snet **[ref]** typed streaming language or the data-parallel functional language Single Assignment C [7], as well as a parallelizing compiler based on the CoSy framework [8] from C to μ TC. In terms of the Microthread Architecture, a μ TC compiler has been developed to produce binaries for this hardware implementation of SVP. Furthermore, there is a serializing μ TC to C verification tool, and finally the μ TC-*p*thread library which is presented and discussed in this paper. Because of this tool chain, μ TC can be seen as an intermediate language, used by the tools to capture concurrency explicitly, but because of the simple and clean syntax extensions, this does not prevent any developer from developing in μ TC directly.

2.3 Microthread architecture

Developed within the Microgrids **[ref]** project, the Microthread Architecture [2] aims to implement a processor which supports the concurrency model defined by the SVP. The concept of the architecture is to design a simple processor which has hardware support for threads and families and their concurrency controls, and that a lot of these processors are integrated together into a massively parallel microgrid chip. A family of threads will in general be run across a group of processors, providing parallel execution.

2.3.1 Microthreaded processor

A Microthreaded Processor consists of an in order, single issue, RISC pipeline, and management hardware to schedule threads. Threads are scheduled in a data-flow manner; only when all data for the next instruction is available will a thread be scheduled. This results in efficient latency hiding on long latency operations like memory reads, by interleaving threads. The processors have a big register file, enough to hold all register contexts of all local threads, and relatively small data and instruction caches.

2.3.2 *Microgrid*

A Microgrid is a CMP implementation using Microthreaded Processors, with special networks to manage the creation, synchronization and termination of families of threads across the chip. It includes an on chip COMA [9] network to implement [10] the asynchronous shared memory model, and a communication network for sending shared variables within groups of processors executing the same family.

3 Motivation

The motivation behind the work presented in this paper is twofold; it provides the first available implementation of SVP, and it is an integral part in the development of an event driven high level simulator to simulate SVP program and system behavior on virtual SVP hardware. In this section we will investigate the possibilities of this implementation, and go into the details of how and why we chose this approach for simulation.

3.1 *SVP implementation*

There are already cycle accurate simulators available for the Microthreaded Processors [ref], which implement the SVP model. However, compilation to these targets currently has to be done by hand or from hand crafted assembly code, as the μ TC core compiler [6] is still under development. Therefore, the library presented here is the first true implementation of SVP which allows μ TC code to run on any machine that has proper *p*thread support; from small embedded processors, to large SMP systems. This is one of the key benefits of this work, as it enables us to develop μ TC programs, execute them, and in that way test and explore the SVP model.

3.1.1 *Framework*

As this implementation bridges the gap between the μ TC and the C++ language using existing standard compilers, it can be used as a framework to aid porting the SVP model to other parallel targets or architectures. The interface to μ TC is there, and the back-end using *p*threads could be modified to for example target the Grid or clusters (e.g. using MPI), or in the opposite direction, to target hardware accelerators in FPGAs. We believe that moving in both directions is possible, but we would have to decompose our concurrency tree into different domains of granularity.

3.1.2 Granularity domains

When targeting other architectures or systems with our library, the difference in granularity of parallelism should be taken into account. Therefore we suggest that the concurrency tree should be decomposed into three granularity domains. The top level families and threads could be distributed over coarse grained parallel environments, for example the Grid or a cluster. The middle level would be distributed on the local nodes from such system, single grid or cluster nodes, SMP machines, and at this granularity *p*threads can be used to distribute and parallelize the work. At the deepest level in the concurrency tree, there might be too much overhead to run in parallel using *p*threads on conventional machines, and threads could be executed sequentially. Furthermore, at the lowest level, executing single threads or families of threads on dedicated FPGA hardware accelerators **[ref]** could be an option.

3.2 SVP simulation

For the development of the high level simulator for various hardware implementations of the SVP model an accurate way of generating traces from μ TC programs was required. This has to be done without losing self-adaptivity, and therefore requires full functionality in the traced program. This is provided by the *p*thread library implementation, and we will go into the background of this in this section.

3.2.1 Simulation requirements

Development of a high level simulator is desired to simulate a large assembly of SANE virtual processors, or alternatively, a Microgrid. As mentioned before, low level cycle accurate simulators are available for a group of Microthreaded processors, which is only single component of a Microgrid. However, these are still isolated from the μ TC perspective, and are too low level to run large system simulations.

There are several reasons why this high level simulator is desired; it can be used to model and investigate the behavior of a self adaptive μ TC program on a large system containing many SVPs, but also research in resource management and protocols on these systems are points of interest. Specifically towards the Microgrid research, interest lies in investigating the load on processors and network interconnections, as well as enabling the development of a resource manager.

3.2.2 Simulator design

Simulation is achieved by co-simulation, splitting it into a fully functional program part that executes the μ TC program and dynamically generates traces by sending events on every special SVP concurrency control. These are then mapped by a discrete event simulator on a hardware model, providing feedback and timing information into the program in order to enable self adaptation. The μ TC part is completely agnostic of time, location of execution, scheduling, load, and only provides the functional part of the simulation. However, it can still receive parameters back from the (simulated) system, for example performance or time, and adapt to this.

3.2.3 Execution methods investigated

Before adopting the *p*thread approach, several other approaches were investigated. Initially a translation of a μ TC program to an equivalent sequential program in standard C was considered. However, the resulting program would lack the ability to self adapt, as there would be no means of parallel monitoring, unless an interleaving scheme would be introduced. Since families of threads are not statically defined and can be created dynamically, this interleaving would become arbitrarily complex.

The current SVP hardware simulators could neither be used for trace generation, as they are low level cycle accurate architecture simulators modeling microthreaded processors, implementing the SVP model in hardware. Compilation from μ TC to this new emerging architecture is still heavily under development, and therefore not suited for the high level simulation.

Interpreting μ TC code was considered as there are open source C interpreters readily available, e.g. CINT that can interpret up to 95% of ANSI C programs [4]. However, these are complex¹, and modifying these to implement SVP instructions and to interpret and keep track of the many parallel contexts present in a μ TC program would likely cost a lot of effort.

After investigating these alternatives, executing μ TC with *p*threads seemed to be the most appropriate choice. Every special SVP instruction in the language had a could be implemented straightforward in library calls that internally call *p*thread functions. All other user code contains classic C, which could be compiled and run natively, allowing support for programs of any complexity. Furthermore, initial experiments have shown that we can run over 90,000 *p*threads simultaneously on a modern Linux 2.6/x86 machine, providing us with enough room to execute complex and heavily parallel μ TC programs using this approach.

¹ The CINT code-base already contains about 80,000 lines of code [4]

4 Implementation

In this section we will go into the implementation details of the library. First we will discuss some general design decisions, then in more detail the implementation of concepts from SVP, and we will look at an overview of the tool chain used to run μ TC programs as *pthreads*.

4.1 Design decisions

4.1.1 C++ vs C

Initially a code transformation from μ TC to C using *pthreads* was investigated, using a translator inserting the additional C with *pthreads* code at every SVP instruction. The first translation was done by hand to test the approach, and was used as a guideline to write the translation rules. However, this approach had several problems; the code would explode by at least a factor of 10, resulting in source code that was hard to read and debug. Additionally, there was a problem to implement shared variables, as they can be used anywhere just as any other type of variable in μ TC, and they are not explicitly read or written. This would require a complete parsing of μ TC to find where these are read for the first time by a thread and where they are written, in order to insert special code to communicate with neighboring threads. Another problem was the *create*, as the thread that is created can have any number of arguments of any type. An alternative approach using C++ solved all these issues, in which we could use the experience and insight we gained of attempting this C implementation.

Using C++ we were able to use templates and classes for a type invariant implementation of shared variables. Furthermore, we could overload its operators, and hide all code to read and write shared variables. As the required code to implement *create* doesn't depend on the number and type of arguments the created thread receives, but only on the parameters to the *create* itself, we could capture this with a template. This resulted in a very clean implementation, where all SVP instructions are calls into library code, and special additional types and classes are defined to implement shared variables, index variables and family identifiers. The syntax offered by this implementation only slightly differs from μ TC, and a simple translator to rewrite the syntax was developed. The resulting translated program is still very readable; its structure is exactly the same as the original in μ TC.

4.1.2 Thread mapping

The implementation uses a one on one approach, mapping every thread in the SVP model onto a *pthread*. This is quite costly because the overhead of creating a new *pthread* is quite large. At a later stage, as suggested in section 3.1.2, sequentialization of threads in the leaf nodes in the concurrency tree can be done to map them onto a single *pthread* for efficiency. This optimization, which has some resemblance to the loop unrolling optimization found in compilers, has not yet been implemented. Further optimizations would include using a pool of threads, so that not every *pthread* has to be created and cleaned up again every time a thread in SVP is.

4.2 Implementation of SVP concepts

4.2.1 Families

As the model works hierarchical, the information about families of threads are stored in nodes which are in a tree hierarchy. Each node in the tree representing a family, with links to its child families, parent families, and its threads. This data-structure is the main infrastructure on which most functions in the library rely, so the family identifier in μ TC is represented by a pointer to the family's node in the family tree.

4.2.2 Shared variables

Shared variables are implemented by defining a template class, as shareds can be any type. In this class the assignment and cast operators are overloaded, implementing the necessary communications to the neighboring threads. A structure in which a shared is stored keeps track of read and write status, as reads are cached, and writes are only allowed once, and to make sure that when a thread terminates and cleans up, it will wait until all shareds have been read by the next thread.

4.2.3 Create

As *create* takes a thread function which can have any number of arguments, *create* has been implemented as a set of template wrapper functions, wrapping around the actual *create* implementation which is independent of the arguments to the threads. At a *create*, a new family object is allocated and put in the family tree, and a *creator* thread is started with *pthread_create* which starts setting up the family and threads in the background, as *create* is not a blocking operation that blocks until all threads have been created. When

the family is set up, this *creator* thread will iterate and use *pthread_create* to start the actual SVP threads in that family, respecting the *block* parameter which controls how the maximum number of threads there should be running simultaneously within the given family. The *creator* thread will exit when either all threads have been created, or the family receives a *kill*, *break*, or *squeeze*.

4.2.4 *Sync*

The *sync* implementation waits for the *creator* thread of the target family to finish, and then checks the family object in the family tree which threads are remaining. Every thread that exits signals this fact through a *pthread* conditional in its family object, and the *sync* will wait on this conditional until it received an exit signal from every thread that was remaining in that family. After this the *sync* reads out the optional return value and return code from the family object, and the family is removed from the tree. When the family the *sync* is waiting on is terminated by a *kill*, *break*, or *squeeze*, a flag in the family object is set and the conditional is signaled to inform *sync* that the family terminated abnormally.

4.2.5 *Kill*

When a *kill* is started, it guarantees using a *pthread* mutex that no new *create* can be initiated, e.g. starting new *creator* threads, as well as no other destructive operations as *break* and *squeeze* or other *kills*. This is to prevent a race between a recursing family and a *kill* trying to recursively terminate it, and races between e.g. *kill* and *break* trying to terminate the same family. After the mutex is obtained the *creator* thread of the target family is signaled to stop creating more threads if it still exists, then the *kill* recurses into all child families repeating the process. At the bottom of the recursion, or after the recursion returns, the family object is checked for all remaining threads and the *kill* sends them a *pthread_cancel*. The *kill* then waits for all threads to exit, removes the family object from the tree and recurses back up, or in case of the top level where it started, it releases the mutex and exits.

The *pthread* cancel state is set to asynchronous when a new thread is created, but the behavior across different *pthread* implementations is not the same. Asynchronous cancellation is defined in the *pthread* standard as to interrupt a thread at “any time”, as where deferred cancellation is defined to interrupt only at certain “cancellation points”. However, the *pthread* implementation on Mac OS X seems to interpret “any time” as “the next cancellation point”, implementing asynchronous cancellation as deferred cancellation. This impacts our implementation as in theory threads could be stuck forever looping in

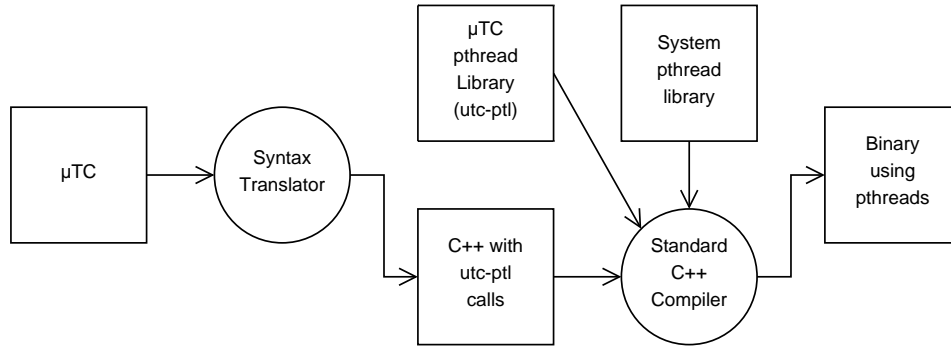


Figure 1. μ TC on *pthread* implementation tool chain

user code without getting canceled, resulting in a *kill* that never completes. This problem is reduced by checking if a family is flagged as being killed when entering some parts of the library, and the assumption that such non interacting looping code is not typical for SVP programs.

4.2.6 *Break*

Break is implemented using the same function as *kill*, with only two additions. First of all it makes sure that the thread that executes the *break* will not receive a *pthread_cancel* from the *kill* function, and second it will set the return value provided in the family object.

4.2.7 *Squeeze*

Squeeze behaves similar to *kill* except that no threads are forced to terminate; it only signals the *creator* thread of the target family to stop creating new threads, and then waits for all threads in that family to complete. Currently our implementation does not yet support recursive *squeeze*, but this will be added soon and will work similar as the recursion in the implementation of *kill*.

4.3 *Tool chain*

The tool chain is formed by a translator that performs the syntax rewrite from μ TC to C++ with calls into the library implementation, which is then compiled with a standard C++ compiler to a binary that runs with *pthread*. An overview of this tool chain is given in Figure 1.

μTC Source	C++ Equivalent
<pre> thread fibonacci_compute (shared int prev, shared int prev2,int *fibonacci) { index i; fibonacci[i] = prev + prev2; prev2 = prev; prev = fibonacci[i]; } thread main(void) { family fid; int fibonacci[10]; int prev, prev2; fibonacci[0] = 0; fibonacci[1] = 1; prev2 = fibonacci[0]; prev = fibonacci[1]; create(fid;;;2;9;1;10;) fibonacci_compute(prev, prev2, fibonacci); sync(fid); } </pre>	<pre> #include "uTC.h" using namespace std; using namespace uTC; void fibonacci_compute (shared<int> prev, shared<int> prev2, int *fibonacci) { uTC::index i; fibonacci[i] = prev + prev2; prev2 = prev; prev = fibonacci[i]; } void _main(void) { family fid; int fibonacci[10]; int prev, prev2; fibonacci[0] = 0; fibonacci[1] = 1; prev2 = fibonacci[0]; prev = fibonacci[1]; create(fid, 0, 0, 2, 9, 1, 10, 0, fibonacci_compute, &prev, &prev2, fibonacci); sync(fid); } </pre>

Figure 2. Syntax translation example

4.3.1 Library

Since the library has to do some initialization in order to implement all the functionality that has been described in detail in the previous section, the library has the main entry point of the executable. After doing initialization it will call the main thread of the original μ TC program. This makes it different from normal software libraries, as it heavily integrates with the μ TC user software, and does not consist of just some function calls. This and the fact that it uses a lot of template functionality makes it impossible to have a binary distribution and dynamically linking library.

4.3.2 Translator

Since the syntax translation is quite simple and trivial, the translator is implemented as a Perl script that uses regular expressions to match and replace the syntax where necessary. Before this is done, some preprocessing using the C preprocessor and GNU indent is done to safely make some assumptions about the formatting of the code. Most of the rewriting is done at the *create* statements, as the named thread and its parameters need to be included into the *create* function-call into the library. Furthermore, the *main* function needs to be renamed, as the entry point moves to the library, and all other named

threads have to be translated to C++ functions, where the definition of shared variables needs to be adapted to the template format. An example of this syntax translation is shown in Figure 2, showing the μ TC code on the left hand side, representing a program that computes the first ten Fibonacci numbers. The translated equivalent C++ code with library calls is shown on the right hand side, still having a very close resemblance to the original program.

5 Results

Our implementation was the first available tool enabling us to execute programs written in the SVP μ TC language, which is indisputably the most important result of the work presented in this paper. In this section we will present some results of multithreading experiments on systems supporting *pthread*s to show that we can create enough *pthread*s in one process to emulate a heavily parallel μ TC program. Furthermore, we will discuss some performance measurements to drive the discussion and conclusion on which granularity this implementation is useful for parallel distribution and not only for emulating SVP or driving simulations.

5.1 *Pthread experiments*

We developed a small testing program to test how many *pthread*s can be created within a single process, which checks if all threads are created and running successfully. After the maximum has been reached, the initial *pthread* stack size is halved and the test is rerun until the maximum number of created threads does not increase anymore. Using this tool we investigated several platforms that were available to us, and we will discuss the results below.

5.1.1 *Linux 2.6*

On a modern Linux 2.6 kernel with the NPTL *pthread* implementation the maximum number of *pthread*s is limited by the value in `/proc/sys/kernel/threads_max`, which has an upper bound limit of the maximum number of processes the scheduler can handle, which is shown in `/proc/sys/kernel/pid_max`. However, `thread_max` is usually set to a lower value by the kernel at boot time, depending on the amount of memory available in the machine. `pid_max` defaults to 32K on 32bit x86 architectures, but can be increased when recompiling the kernel.

As these limits can be increased, the real limit of the number of threads

that could be created was imposed by the amount of virtual address space a process gets assigned. Each thread requires some address space for a stack, and on a 32bit architecture there is 3GB address space available for a process under Linux. A 32K stack seemed to be the minimum, which allowed us to successfully run 91,841 threads simultaneously, after tweaking our Linux kernel in order to increase *pid_max* and *threads_max*. On a true 64bit architecture where more virtual address space is available, these limits will likely be orders of magnitude higher.

5.1.2 Solaris

We have run the same test on Solaris 8 and 9, on Sun Ultrasparc machines. On Solaris 8 we could create around 3395 threads, and on Solaris 9 around 4075 threads. However, these numbers seemed to be influenced to the number of processes running under that user, and we did not have the ability to configure the parameters of the systems.

5.1.3 Mac OS X

We have experimented with Mac OS X 10.4 (*Tiger*) and 10.5 (*Leopard*), and on both PowerPC G5 and Intel platforms we could create up to 2560 threads per process. However, experiments revealed that when the maximum number of threads is reached on Mac OS X 10.4, the *pthread_create* call does not return an error as it should, and acts as if a thread has been created while it has not. This problem is not present in the 10.5 release.

5.2 Measuring overhead

We have measured the time it takes to create a family of threads on the platforms available to us, to get an idea of the overhead induced by the library. The results are shown in Figure 3, where the time has been normalized to the estimated number of clock cycles to compare the different systems, and the results are shown for families of independent threads and families passing one shared variable to measure the difference in overhead of communicating shared.

As expected, the overhead increases linearly as larger families of threads are created, as the *creator* thread as explained in 4.2.3 iterates and creates the threads. For Mac OS X initially setting up a family takes in the order of $30 \cdot 10^4$ cycles, and additionally around $20 \cdot 10^4$ for every thread, and $3 \cdot 10^4$ for a shared variable. Solaris takes between $7 \cdot 10^4$ and $14 \cdot 10^4$ cycles to set up a family, $10 \cdot 10^4$ for every thread, and $5 \cdot 10^4$ cycles for a shared variable.

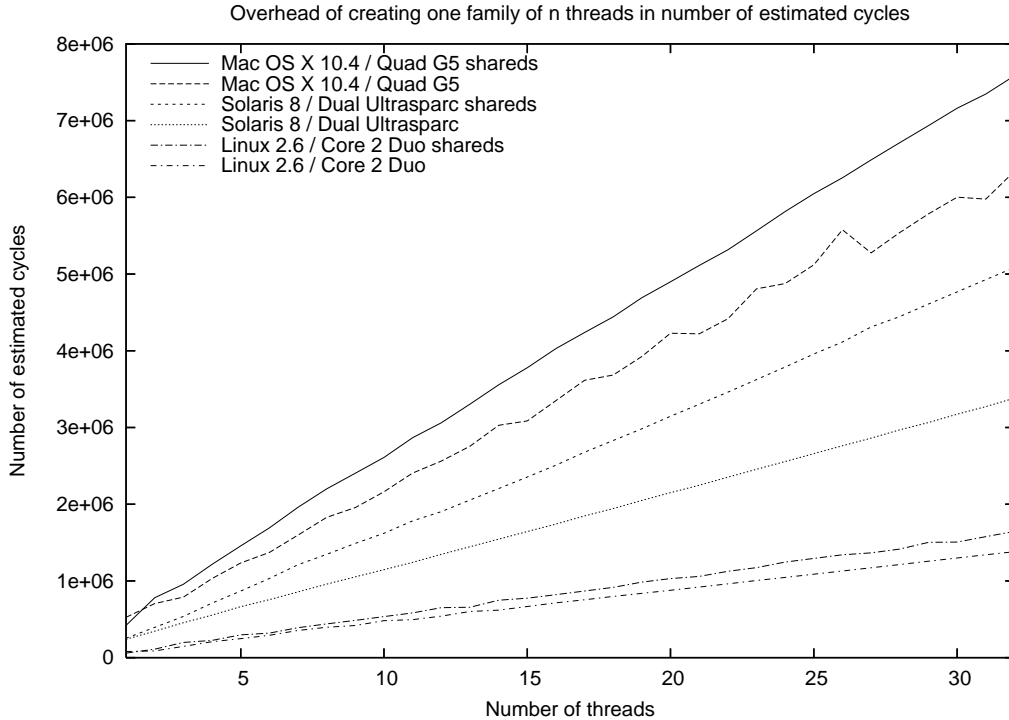


Figure 3. Estimated overhead in clock cycles of creating one family of n threads on several platforms and architectures, with and without the use of shared variables

Linux sets up a family in $3 \cdot 10^4$ to $4 \cdot 10^4$ cycles, $4 \cdot 10^4$ cycles for every thread, and 10^4 additional cycles per shared. These numbers show that the order of magnitude of overhead created by the library is in the 10 to 100 thousands of cycles, and when using this implementation as a basis to implement SVP, this is the kind of granularity at which domains should be separated as explained in 3.1.2.

5.3 Measuring speedup

We now have an idea of the order of magnitude of the overhead from the previous section, and will now investigate the behavior when executing and distributing a simulated workload on multiprocessor systems. We have experimented with dividing a simulated workload in the order of 10^7 operations over a family with a varying number of threads. The results are shown in Figure 4, and show one run with independent threads, and one experiment where the threads communicate through a shared variable halfway the workload for each system.

The experiments were run on two two-way and one four-way system, and for a low number of threads we see that the expected speedup is obtained. For the four processor G5 system the speedup converges towards 3.5, where the



Figure 4. Speedup on simulated workload executed by one family of n threads on several platforms and architectures, with and without the use of shared variables

sawtooth behavior in the graph can be explained by the uneven distribution of threads over the processors when they are not a multiple of four. Also the difference between the runs with and without shares show that when the threads are more synchronized they are better distributed over the processors by the system. Similar behavior can be observed at the graphs representing the dual Ultrasparc system, where the speedup converges around 1.9, and the Core 2 Duo system where the average speedup is around 1.7. The behavior between 28 to 30 threads on the Linux / Core 2 Duo system could have something to do with the memory or cache system, as it is not a true dual CPU but a dual core machine.

6 Conclusion and Future Work

In this paper we have presented our work on implementing SVP represented by μ TC on conventional systems using POSIX threads. We have shown that it is a viable approach as the investigated platforms had no problem supporting thousands of threads running within a single process, and we have investigated the scalability and overhead of our implementation. The motivation behind this work and the possibilities of using this work as a framework have been explained, and we will now give an overview of the future work and possible future directions.

6.1 Optimizations and porting

A lot of work can still be done in optimizing the implementation, for example by using a pool of p threads to reduce the overhead of thread creation, and tweaking functionality in the library to the specific target platform or architecture. Furthermore, porting to different granularities of architectures as mentioned in 3.1.2 needs to be investigated. In order to port this implementation to Grid or cluster systems, a solution has to be found for the lack of shared memory in such platforms. One way to do this is would be by defining in- and output memory ranges for a family, and sending the inputs at creation and receiving back the outputs at *sync*. For such a port also the sequentialization at low level threads might be required when the granularity gets down to the overhead of creating threads. This could be implemented by modifying the *creator* thread to iterate and execute the thread functions one after another instead of creating them in parallel as new p threads.

Another interesting target to port this work to, as mentioned before, are FPGA platforms with hardware accelerators. Work has already been done to port the software to Xilinx Microkernel[13] on Microblaze[12], and we hope to work on a port that includes accelerated threads or families of threads on the master/worker platform[**ref**] developed at UTIA [11].

6.2 Simulation

The other side of the future work is on the development of the high level SVP system simulator. The interfaces still have to be clearly defined, and the simulator has to be built. Furthermore, the syntax translator could be modified to automatically add code annotation to abstract the workload induced by code between SVP instructions. Schemes could be developed to abstract and parameterize families of threads, in order to improve simulation performance.

References

- [1] C.R. Jesshope, SVP and μ TC - A dynamic model of concurrency and its implementation as a compiler target, <http://staff.science.uva.nl/~jesshope/Papers/uTC-paper.pdf>
- [2] C.R. Jesshope, Microthreading a model for distributed instruction-level concurrency, *Parallel processing Letters*, 16(2), pp209-228, ISSN: 0129-6264
- [3] C. R. Jesshope, μ TC - an intermediate language for programming chip multiprocessors, *Proc. Pacific Computer Systems Architecture Conference 2006*

- ACSAC06, ISBN 3-540-4005, LNCS 4186, pp147- 160

- [4] CERN, The CINT C/C++ Interpreter project homepage, <http://root.cern.ch/twiki/bin/view/ROOT/CINT>
- [5] AETHER project homepage, <http://www.aether-ist.org>
- [6] T.A.M. Bernard, C.R. Jesshope, and P.M.W. Knijnenburg, Strategies for Compiling μ TC to Novel Chip Multiprocessors, International Symposium on Systems, Architectures, MOdeling and Simulation, S. Vassiliadis et al. (Eds.): SAMOS 2007, LNCS 4599, pp. 127-138, 2007
- [7] S.B. Scholz, Single Assignment C - Efficient Support for High-Level Array Operations in a Functional Setting, Journal of Functional Programming, 13, (6) (2003) 1005-1053
- [8] ACE: CoSy Compiler Development System, <http://www.ace.nl/compiler/cosy.html>
- [9] E. Hagersten, A Landin and S. Haridi, (1992) DDM - a cache-only memory architecture, IEEE Computer, 25(9), pp. 44-54
- [10] L. Zhang, C.R. Jesshope, On-chip COMA Cache-coherence protocol for Microgrids of Microthreaded Cores, presented at and submitted to Proc. Workshop on Highly Parallel Processing on a Chip (HPPC), associated with Euro-Par 2007
- [11] UTIA website, <http://www.utia.cas.cz>
- [12] Xilinx, "MicroBlaze Processor Reference Guide", 2007
- [13] Xilinx, "Xilkernel" Manual, 2006