Enabling Task Level Parallelism in HandelC

Thamer S. AbuYasin

Submitted to the Department of Electrical Engineering & Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Thesis Committee:

Dr. David Andrews: Chairperson

Dr. Arvin Agah

Dr. Perry Alexander

Date Defended

2007/12/14

The Thesis Committee for Thamer S. AbuYasin certifies
That this is the approved version of the following thesis:

Enabling Task Level Parallelism in HandelC

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Chapter 1

Abstract

HandelC is a programming language used to target hardware and is similar in syntax to ANSI-C. HandelC offers constructs that allow programmers to express instruction level parallelism. Also, HandelC offers primitives that allow task level parallelism. However, HandelC does not offer any runtime support that enables programmers to express task level parallelism efficiently. This thesis discusses this issue and suggests a support library called HCthreads as a solution. HCthreads offers a subset of Pthreads functionality and interface relevant to the HandelC environment. This study offers means to identify the best configuration of HCthreads to achieve the highest speedups in real systems.

This thesis investigates the issue of integrating HandelC within platforms not supported by Celoxica. A support library is implemented to solve this issue by utilizing the high level abstractions offered by Hthreads. This support library abstracts away any HWTI specific synchronization making the coding experience quite close to software.

HCthreads is proven effective and generic for various algorithms with different threading behaviors. HCthreads is an adequate method to implement recursive algorithms even if no task level parallelism is warranted. Not only HCthreads offers such versatility, it achieves modest speedups over instruction level parallelism ad-hoc approaches. The Hthreads support library served its intended purpose by allowing HCthreads real system tests to proceed on a third party platform. No major issues were reported while conducting these tests, still additional investigation and verification is required.

Contents

List of Figures

List of Tables

Chapter 2

Introduction

HandelC is a programming language and a hardware description language that offers constructs which allow the programmer to express mainly instruction level parallelism [9]. Nevertheless, these constructs can be used to express higher levels of parallelism in a primitive manner. Programmers should come up with their own framework to express higher levels of parallelism using these constructs. It is expected for programmers to come up with different TLP solutions to solve for different algorithms some efficient and some not. HCthreads was intended to address this issue in detail and provide a global and an acceptable solution.

HCthreads is based on POSIX threads [6]. Pthreads is now a familiar thread library with a widening base of programmers. The POSIX standard defines the API or policy and not the implementation of the underlying middle layer, enabling portability across various platforms. It would be logical to assume that this knowledge will be advantageous when modeling HCthreads after Pthreads. On the other hand, given the makeup of hardware in general and HandelC in particular HCthreads should be light weight so it does not inflect additional costs in performance. This can be achieved by adapting the representation and seman-

tics of the proposed threading interface so all needs are covered adequately and without surplus. Also, a fine tuning framework is described to help developers minimize resources used by HCthreads as much as possible. This is done by profiling the library when operating on a certain application. The profiling results might indicate the need to expand or even shrink the library to achieve the highest speedups in real systems.

This study also addresses the issue of integrating HandelC cores within platforms not supported by Celoxica. The solution was to utilize the preexisting Hthreads system and its distinction in bridging the HW/SW boundary [4]. In this approach HandelC code resides as user logic inside a hardware thread, therefore it completely benefits from all services offered by the Hthreads system and the HWTI.

The remainder of this thesis is organized as follows. First, the value of task level parallelism over instruction level parallelism is discussed and then the contributions of this thesis are listed. Second, a brief background of FPGAs, related works and the HandelC language is presented. The major features of Pthreads and Hthreads are then discussed.

A detailed description of the HCthreads attributes, components, interface, data structures and the implementation aspects of the threading library is then presented. The Hthreads support library and the fine tuning framework are then discussed.

Finally, an experiment setup, presentation and analysis of results are presented showing correctness, performance and utility. The conclusion is then presented along with proposed future work.

Chapter 3

Problem Statement

The first section in this chapter tries to make the case against instruction level parallelism. The history of ILP is investigated and its shortcomings are presented. The justification for task level parallelism is then made and research efforts relating to the topic are presented. Finally, the contributions of this thesis are listed in the last section.

3.1 Instruction Level Parallelism vs. Task Level Parallelism

For the past decade and a half a lot of research was conducted trying to exploit FPGA flexibility to achieve speedups when compared to general purpose processors.

The first notable effort in this field was the proposed PRISM system by Athanas and Silverman [5]. PRISM was a HW/SW system that operated on the assumption that certain computationally intensive algorithms spend most of their execution time inside a small portion of the code. The system extracted fre-

quently accessed code blocks at compile time and synthesized them into hardware images. On the other hand, the software images running on the processor would include new custom instructions responsible for coordinating with the accelerated code blocks on the FPGA.

The second major work in this field was the DISC system by Wirthlin and Hutchings [27]. DISC follows the same concepts defined by PRISM with one major difference. Instead of limiting the reconfigurable fabric to one custom instruction DISC maintains a collection of synthesized custom instructions that can be paged in and out of the FPGA using runtime reconfiguration.

The two systems mentioned above rely on the concept of instruction set metamorphosis to prove the utility of HW/SW co-design. The synthesized hardware in these two systems exploited ILP at a basic level to achieve speedups. On the other hand, TLP was not an option due to the limited reconfigurable fabric available in that period.

Then GARP, by Callahan and Wawrzynekth, is one of the first research efforts to directly address ILP [8] [7]. GARP borrowed techniques from VLIW processors to indentify loops and exploit what was believed to be large degrees of ILP in sequential code. GARP employs various techniques such as speculative execution, pipelining, data dependency resolution and others to achieve ILP. However, the overhead of synchronizing between the processor and the GARP array was larger than the gains in most cases. There wasn't much ILP for the co-processor to work on.

SPARK is another major work, by a team of researchers from UCI, that deals with ILP [14]. The major objective of SPARK is to translate behavioral descriptions from ANSI-C to synthesizable VHDL. SPARK employs a lot of techniques

to enhance ILP such as scheduling, loop transformations and speculative code motions. SPARK shows some decent results with regard to area and total delay but no results with respect to total execution times and actual speedups are presented.

There are other studies that extract ILP with new techniques such as DEFACTO and ASC [10] [17]. Regardless, these studies show that there is little ILP to automatically or manually extract [26]. Likewise, ILP in general purpose processors, through superscalar processing and pipelining, has reached it limits. However, the field of parallel computing as well as the advent of CMPs and Hyper-Threading helped show that performance gains in TLP are an order of magnitude greater than gains attained with ILP alone [23]. Therefore, FPGAs can benefit from TLP because they are flexible to adopt any computational model. Yet it is important to mention that TLP is not an alternative but a supplement to ILP, both explore parallelism in a different scope hence employing both will increase possible performance gains even more.

In the venue of TLP and FPGAs four research efforts stand out. These works were originally proposed to bridge the HW/SW boundary. Each outlined a MIMD computational model and consequently a programming model that abstracts communication and synchronization between threads running in hardware and software.

Miljan in EPFL achieved this goal by expanding the concept of virtual memory to hardware platforms [24]. Platzner of Paderborn proposed a system called ReconOS to reach this objective, particularly ReconOS offers runtime reconfiguration support [16]. Vahid in UC Riverside constructed a simulation environment that dynamically translates binaries to run on reconfigurable fabric [22]. Also,

Andrews et al. from the University of Kansas reached the same goal by adopting the Pthreads programming model and by implementing the required middleware directly into the reconfigurable fabric [3]. This expanded their targeted customer base to include real time embedded systems developers [1].

These approaches enable the developer to include different numbers of hardware and software threads, thus providing the FPGA audience with the desired TLP. Unfortunately, these studies do not reflect on the TLP gains that might be achieved. Most test cases and experimental results carried out with these systems were to assert the validity of the model and the inflected overhead.

3.2 Thesis Statement and Contributions

The major objective of this thesis is to expand the capabilities of the HandelC language by enabling TLP alongside native ILP techniques. HandelC has a large domain of users that might benefit from such enhancement. This is done by constructing a simple library that allows HandelC developers to express parallel tasks easily. This library employs only HandelC constructs thus eliminating additional verification work with components outside the language boundary. The library is fast enough to reduce the execution overhead over ILP ad-hoc approaches. The library does not incur severe performance overheads in terms of area and timing when compared to these approaches. The library defines a framework to allow the prediction of utilization patterns of the library recourses. This framework helps developers identify whether to increase or decrease the number of instantiated components thus improving overall system performance.

The other goal of this work is to enable the integration of HandelC implementations within systems not supported by Celoxica. This is done by exploiting the

abstractions offered by Hthreads and the HWTI thus facilitating communication and synchronization across the HW/SW boundary. This support library abstracts away HWTI specific operations and signals such that the written HandelC code resembles software.

Chapter 4

Background & Related Work

In this chapter a brief background of FPGAs is presented in the first section. Second, related works in the field of C to Hardware translation are listed. Third, details of the HandelC language are discussed. Aspects of Pthreads along Hthreads are then presented in the last section.

4.1 Field Programmable Gate Arrays

The story of FPGAs started with Estrin and the well-known Pasta's challenge in 1959 [12]. Then, Estrin proposed the variable plus fixed architecture, which denotes the union between hardware and software systems. Limited by the technology available in that time period, that novel system had to wait a couple of decades before becoming a reality. Nowadays, FPGAs are taken for granted and are becoming popular in various fields such as high performance computing, networking, image processing and hardware prototyping.

FPGAs are made of a large collection of CLBs or configurable logic blocks. Each CLB is made of a look up table and other combinational and synchronous

components. The lookup table allows the CLB to assume different Boolean functions according to what is stored in its memory. A large collection of these CLBs interconnected together builds a more complex digital circuit. FPGAs in practice are made of such large collection of CLBs plus other specialized components such as multipliers and general purpose processors. FPGA vendors, such as Xilinx and Altera, use these principles in building their different product lines.

4.2 C to Hardware Translation

Section 3.1 went through some techniques that exploit ILP by transforming ANSI-C code into hardware. GARP, DEFACTO, SPARK and ROCCC [13] are research efforts that fall under this umbrella. These systems capitalize on ILP in the transformed code to establish a SIMD model from the original SISD code. This is the most common tactic when it comes to C to Hardware translation. Other methods construct either a heavily pipelined SISD or MIMD circuits [2]. These transformation tools require the programmer to use some pragmas to guide the translation process. Also, these tools leave out important aspects of the ANSI-C language such as pointer support and recursion. These tools were mainly constructed for academic investigation and they still fail in abstracting the HW/SW boundary effectively.

On the other hand, some commercial tools such as HandelC, ImpulseC, NapaC and DimeC are available with a higher success in achieving favorable C to Hardware transformations on the one hand and HW/SW integration on the other hand. These tools require the user to employ additional pragmas to guide the translation process. Also, these tools limit the programmer in the set of supported ANSI-C constructs that can be migrated to hardware. Edwards in his famous article [11]

sifts through these languages indentifying the weaknesses and the strengths of entire C to Hardware movement, his final conclusion is why bother, which is not without merit.

4.3 The HandelC Language

HandelC is a language based on the Communicating Sequential Processes algebra and its embodiment in Occam [19]. HandelC was developed by researchers in Oxford University, England and is currently maintained by Celoxica.

HandelC syntax is based on conventional C syntax and the language targets low level hardware. Because HandelC is based on C syntax it is essentially sequential. However, in order for the language to reap the full benefits of hardware and the massive performance increases it offers, it is important to use a parallel programming model. HandelC provides native constructs that instructs the compiler to build and execute statements in parallel supporting SIMD operations.

The main characteristic of HandelC forces assignments to happen in one clock cycle. This restriction drives all expressions and control constructs in the language to be implemented in pure combinational circuits so it can be evaluated instantly. This feature makes the language cycle accurate. This fact insures that simulations of implemented programs correctly map the behavior of hardware, consequently reducing the development cycle considerably. Unfortunately this comes at a price, in certain cases the combinational logic will become deep to the point where timing constraints cannot be met.

When it comes to pointers HandelC is more flexible than other C to Hardware translation tools. HandelC allows the use of pointer arithmetic, the indirection operator, the address of operator and even pointers to functions. However, some

restrictions exist such as limits on casting, pointer comparison with each pointing at a different array. Still, the main issue is the inability to allocate resources dynamically, pointers may only hold valid addresses of preallocated registers at compile time. However, this restriction is expected in hardware where there is no free store by default.

HandelC offers different ways to express code encapsulation such as shared functions, inlined functions and macro procedures. These methods have different mappings to hardware. Some offer typed return values, some offer shared hardware and others offer multiple copies. These constructs allow the programmer to choose the best way to express the current functionality depending on the available resources and the timing requirements.

HandelC allows the programmer to implement more than one main function. This allows the programmer to have independent threads of execution. These mains can share the same global scope and synchronizing between them can be achieved by the use of spinning semaphores. Different mains in HandelC can be declared under different clock domains. Though this will add a penalty when trying to synchronize or communicate between these different clock domains, certain performance increases can be accomplished.

Channels are another important feature in HandelC. These abstract data types are used primarily to communicate between parallel blocks of execution. Each channel connects between two communication partners; any end will be halted till the other end in the communication pair is ready. This behavior allows programmers to use channels as synchronization primitives between independent threads of execution.

HandelC channels, spinning semaphores and parallel mains allow the program-

mer to implement TLP statically. However, the language does not offer runtime support for TLP which is counter intuitive to programmers in general.

4.4 Parallel Programming Models

4.4.1 Pthreads

Threads can be used to implement parallelism in shared memory architectures. Historically, hardware vendors have implemented their own proprietary versions of threads making portability an issue for programmers. This was solved with the arrival of the POSIX standards. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads [6].

Pthreads is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on multiple processors thus gaining speedups through parallel or distributed processing. Gains are also found on single processor systems which exploit latency in I/O and other system functions which may halt process execution.

Pthreads provides spinning locks and blocking mutexes as the basic synchronization primitives. Also, Pthreads offers conditional variables, broadcasts, timeouts and signals for better and more intuitive synchronization. Creating a thread in Pthreads is done by calling the pthread_create function passing a pointer to the function to be executed and a pointer to the input structure. Joining on a thread is done by calling pthread_join and choosing which thread to join with, or the programmer can join on threads by using mutexes and conditional variables. Threads can be in the detached state by either setting the attributes when cre-

ated or calling the function thread_detach. The functionality of Pthreads is quite simple and well known to programmers.

4.4.2 Hthreads

Hthreads is a real-time operating system designed and implemented in the University of Kansas and is compatible with Pthreads [15]. Although Hthreads was designed and built under the idea of HW/SW co-design, the system enables programmers to implement multiple threads in hardware and software seamlessly. Hthreads can achieve true physical concurrency by enabling multiple hardware threads to work hand in hand with software threads. Without Hthreads or similar systems programming threads or parallel tasks in hardware is a difficult endeavor with a very high initial investment for every new algorithm.

The major components of Hthreads are implemented in hardware yielding faster scheduling cycles and fewer interruptions to the CPU. This will minimize the number of context switches the CPU has to do. Also, Hthreads offers low jitter and more deterministic execution times. It is worthy to mention that Hthreads was the inspiration behind HCthreads.

Chapter 5

HCthreads: Design, Architecture & Implementation

It was decided that a subset of the Pthreads interface is needed in HCthreads. Pthreads APIs that deal with attribute maintenance are redundant in HCthreads. Pthreads was intended to serve a large range of applications from enterprise solutions to simple programs. However, HCthreads is meant to facilitate achieving true physical concurrency for homogeneous threads on a limited platform, there is no room for diversity. Hence, HCthreads was built to accommodate this fact by expressing some attributes as compilation flags resolved at design time. This helped slim down the interface, thusly conserving hardware as much as possible.

The other main segments of Pthreads not implemented in HCthreads are conditional variables and blocking mutexes. Conditional variables and blocking mutexes are essential in achieving pseudo concurrency in a single processor platform. In Pthreads, these synchronization primitives are the means programmers employ to indicate which thread should be on the processor in a certain time period and which should not. Schedulers depend on such a programming model to decide

which thread should be active and which should be dormant [21].

HandelC however provides spinning semaphores as a native construct in the language, there is no need for blocking mutexes. In single processor platforms there is no sense for a programmer to use a spinning mutex. The processor will be occupied by the thread trying to lock the spinning mutex, no other thread can run. Concurrency here can only be achieved through time slicing, thus a blocking mutex is necessary. Also, in multiple processor platforms a thread trying to lock a spinning semaphore for a long period of time will generate a lot of bus traffic. The only justification for spinning semaphores in multi processor shared memory architectures is when the spinning mutex can be acquired faster than a context switch [21].

Fortunately that is not the case in HandelC. In HandelC and FPGAs concurrency is physical, functional units are duplicated and not shared. If one functional unit is trying to get a hold of a spinning semaphore the others will be performing their assigned tasks regardless. Also, HandelC does not employ a bus structure internally. Thus, blocking on a spinning semaphore will not produce any overhead. On the other hand, context switching has no meaning in HCthreads, thus implementing blocking mutexes is not possible. In HCthreads functional units are main threads of execution responsible for synchronizing threads as function calls to hardware resources. After invocation, these functions can only exit when they return. There is no notion of an instruction stream like in the Von Neumann architecture, and there is no notion of state like in Hthreads. Nevertheless, the concept of context switching along with state, conditional variables and blocking mutexes can be added to HCthreads but with a cost that can never be justified to both the programming model and FPGA resources.

In this chapter HCthreads attributes such as joinable threads versus detachable threads are discussed in the first section. Second, the system main components are presented. Third, the simple HCthreads interface is described. The HCthreads data structures are then detailed in the last section.

5.1 HCthreads Attributes

Stand alone FPGAs are used to implement simple systems that may be divided to homogeneous parallel tasks. These devices are not suitable for large scale computer systems with heterogeneous parallel tasks. This assumption comes from the original purpose FPGAs were made for. FPGAs were constructed for certain applications such as high performance computing and embedded systems rather than being the platform of choice for generic information systems [12] [25]. Therefore, Pthreads attributes for individual threads were extracted to a collection of compilation flags in HCthreads. These flags will force all threads in that particular system to expose the same behavior. Adopting this strategy helped save a considerable amount of hardware and improved timing.

5.1.1 DETACHED

pthread_detach was replaced by a compilation flag in HCthreads to conserve hardware. Otherwise, the state registers for threads would have expanded and extra logic would have been implemented to accommodate a joinable-detachable scheme in the same design. Instead, a compilation flag defined by the programmer at design time sets the intended behavior of all threads in the system. It is advisable to implement a detachable threading strategy rather than a joinable one due to the extra hardware and synchronization joinable designs require. Please

check Section 5.3.

5.1.2 CONTAINER_SIZE

The default value for the flag CONTAINER_SIZE in HCthreads is thirty two entries. Though this size is quiet sufficient for most algorithms, the programmer still can increase or decrease the number of entries in the ready to run container. The programmer needs to define and assign a value to the compilation flag CONTAINER_SIZE before including the HCthreads header file. CONTAINER_SIZE should assume a value according to the following equation:

$$CONTAINER\_SIZE = 2^X, \tag{5.1}$$

Where X ranges from 2 to 6. There is no point in having queues with 1, 2 or more than 64 entries. The reason for this restriction is the simplicity of implementing queues using overflow in hardware. The ready to run container might be required to behave like a queue. Then, it is easier to implement the bound markers using over flow rather than adding extra logic.

5.1.3 R2RSTACK

The ready to run container by default exposes a queue behavior. However, in certain algorithms, such as NQueens, it is wiser to make the ready to run container expose a stack behavior. In these algorithms each thread will spawn a good number of threads before terminating. If the queue behavior of the ready to run container was kept, HCthreads might run into the limitation of a breadth first search [20]. Every thread will spawn enough threads that will fill up the ready to run container. Then the current executing threads are still occupying

the functional units unable to terminate because they still need to spawn more threads. On the other hand, making the ready to run container expose a stack behavior will force the threads to behave quite close to a depth first search solving this problem.

5.1.4 NO_FNUNITS

This flag specifies the number of functional units the current implementation of HCthreads contains. Different algorithms require different optimal numbers of cores running concurrently to improve overall performance. This is governed by memory access patterns and the physical limitations of the targeted platform. The designer is provided with a framework to assess the need for such resources as described in Section 6.3.

5.2 HCthreads Components

The three HCthreads components are main functions other than main thread of execution. HandelC allows programmers to implement any number of main functions running independently from each other. HandelC provides channels and spinning semaphores so programmers can synchronize between these mains.

5.2.1 The Dispatcher

This is a very light weight component that assigns threads to free functional units and at the same time signals these functional units to start executing their assigned threads. This component also initializes all state registers in HCthreads during the first clock cycle when the system starts. The dispatcher after that enters an infinite loop where every clock cycle it keeps looking for threads to

be scheduled by checking the availability of free functional units and threads in the ready to run container. Once a scheduling decision is made the global synchronization semaphore is locked and in the case or a ready to run stack in three clock cycles a thread is assigned to a functional unit. Otherwise, in the case of a ready to run queue a thread is assigned to a functional unit in two clock cycles. The reason for this discrepancy is the difference in data dependency of the ready to run container markers when behaving like a stack or a queue. During the last clock cycle in a successful scheduling event the thread in question starts to execute. This is done by sending a pointer of its argument from the thread argument array via a HandelC channel. This will release the other end of the communication channel in the free functional unit identified during the previous clock cycle. These operations are carried out using simple combinational logic including one priority encoder to retrieve the free functional unit, shift and bitwise logic operations, a bunch of variable assignment and channel write statements.

5.2.2 The Terminator

The first implementation of the terminator is a separate component that works with the other end of functional units. This implementation utilizes a special construct in HandelC called the PriAlt statement. The PriAlt statement is control statement constructed just like the switch statement. However, instead of branching to different segments of code depending on the value of the control variable, the branching will happen according to the readiness of a communication channel with one end being in the case list inside the PriAlt statement. When two communication channels are ready at the same time the arbitration scheme necessitates that the transaction to be completed first is the communication case listed first,

the other one would be blocked until the next pass. In case of no ready communication channels, the PriAlt statement would block till one communication case is ready. This implementation keeps HCthreads data structure manipulation for thread exit in one central location. In this design the terminator will listen to all functional units at the same time and once a functional unit signals completion its id is saved. Then the global synchronization semaphore is locked and the state of the functional unit in question is set to free and the finished thread is marked joinable under a joinable design. In this design a single termination round takes four clock cycles to execute in case of a joinable scheme and three clock cycles to execute in case of a detachable scheme. Unfortunately, in certain instances this design cannot simulate due to a known bug but can execute in hardware, therefore another supplementary implementation was proposed.

The second implementation is to take the termination logic and duplicate it at the end of each functional unit. This will eliminate the need for a separate parallel main function to act as a terminator. Also, this implementation will get rid of the second set of communication channels responsible for synchronizing all functional units with central terminator. In this design a single termination round takes two clock cycles in case of a joinable scheme and one clock cycles to execute in case of a detachable scheme. This design is used only in simulation when the first design fails; this design was never used in a the real system tests.

5.2.3 The Functional Units

These are a bunch of identical main functions each responsible for executing a separate copy of the function the developer wishes to thread. Each of these functions is implemented as an infinite loop where HandelC communication channels

maintain synchronization between thread calls and the dispatcher from one end and the terminator from the other end depending on the terminator design. If the dispatcher decides to schedule a thread on a certain functional unit, it will signal that functional unit by sending a pointer to the input structure of that thread using a communication channel. After that, the threaded function on that functional unit starts to execute and upon completion the functional unit will do one of the following. The functional unit in the first implementation of the terminator will signal that component using a communication channel communicating its id. After that, the functional unit will block again on the first channel waiting for the dispatcher to schedule another thread. The termination logic in this case is carried out inside the terminator component. On the other hand, the second terminator implementation requires the termination logic to be implemented after the thread call. The functional unit itself is responsible of locking the global synchronization semaphore and properly updating the corresponding data structures.

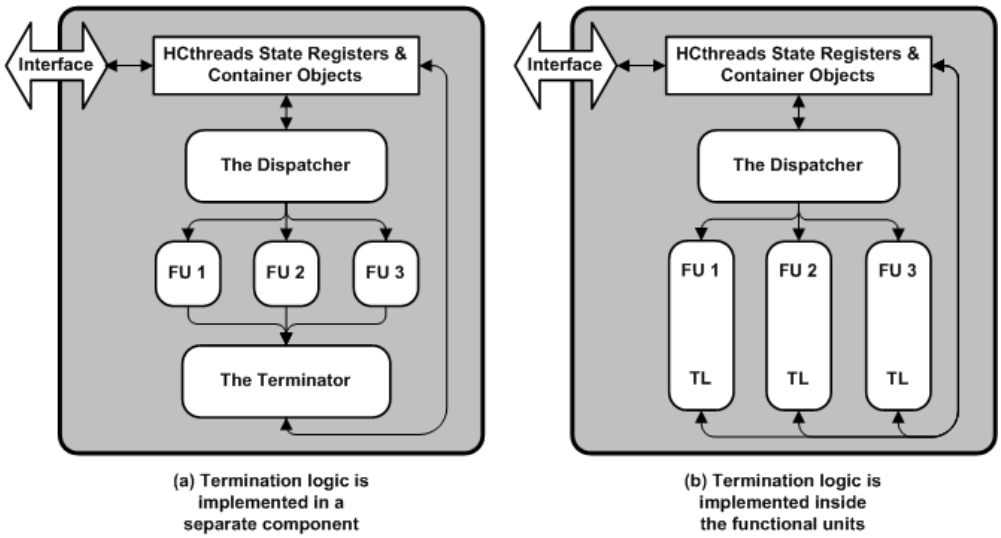

Figure 5.1. HCthreads internals assuming three functional units

5.3 The HCthreads Interface

To keep things simple HCthreads allows the programmers to call two functions only, hcthread_create and hcthread_join. These two functions behave like their Pthreads counterparts, but they have different decelerations, signatures and return values. hcthread_join accepts only a thread id by value and halts the execution of the current main function till the corresponding thread terminates. hcthread_join does not have a return value; the thread will either eventually join or halt for eternity. However, joining on a valid thread id is guaranteed not to halt giving that the thread in question is going actually to terminate and not halt due to congestions in the functional units and/or the ready to run container. It is recommended not to use a joinable thread scheme, it might generate deeper control flow and worse timing scores, instead check Sections 7.2 7.3.

hcthread_create was reduced to two inputs in the case of a joinable design and only one input in case of a detachable design. For joinable designs the thread id by reference and the input structure by value are needed. For detachable designs the input structure by value is needed. The thread id section is still common with pthread_create, but it was kept like that to maintain compatibility with hcthread_join should the programmer choose to. Unlike in Pthreads the input structure is passed by value due to HandelC pointer restrictions, for more information refer to Sections 5.2. hcthread_create needs to inform the programmer if a thread was successfully created for the current call or not, so the return value is binary, there is no other information to submit back.

Both hcthread_create and hcthread_join are inline functions. HandelC programmers might call different copies of these functions concurrently. To achieve better parallelism different hardware copies of these functions should be respon-

sible for executing different invocations.

The hcthread_create function is not expensive given that the function, after locking the global synchronization semaphore, takes only two clock cycles if successful and one clock cycles if not. The function contains one if statement with no logic operators, one priority encoder, shift and bitwise logic operations and a handful of variable assignments. The hcthead_join function in joinable designs contains one while loop that locks the current thread of execution till the corresponding thread is in a joinable state, that is done before locking the global synchronization semaphore. Once the thread in question is ready to be joined, the global semaphore is locked and in one clock cycle the state registers are updated using shift and bitwise logic operations.

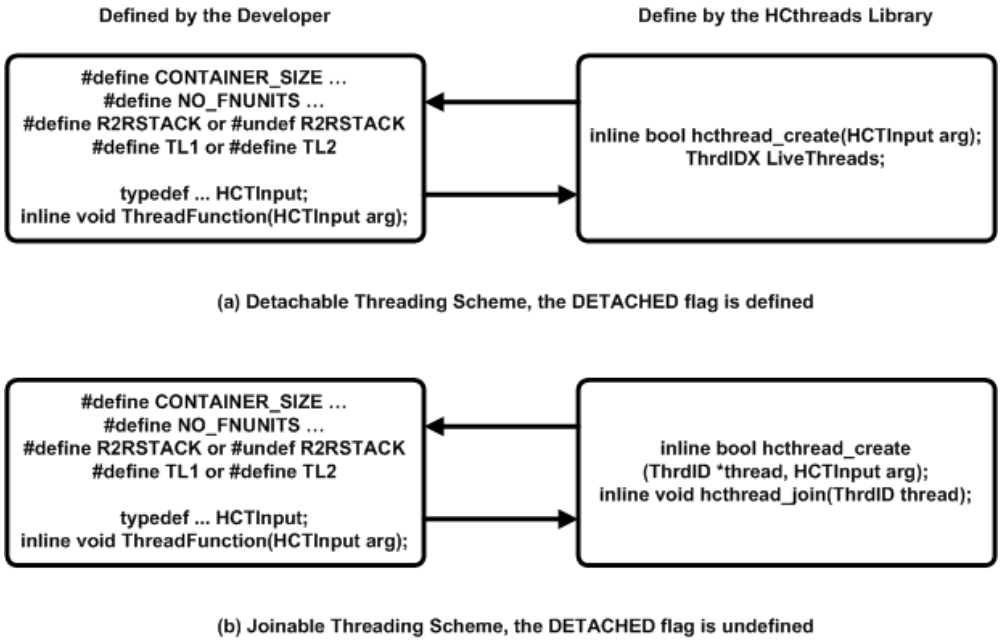

Figure 5.2. The HCthreads interface as defined by the developer and the library

To complete the interface the developer is supposed to define the input struc-

ture and a prototype of the function to be accelerated according to specific names and signatures. Due to limitations in the HandelC language and the difference in pointer semantics between hardware and software, these two components cannot be supplied at run time via pointers. Both should be statically laid out at design time before including the HCthreads library so the library can preallocate resources for its data structures and functional units before compilation.

5.4 HCthreads Data Structures

The HCthreads library employs few structures that maintain state for the threads and the functional units in the system. These structures are the ready to run container, its supporting markers, the arguments array, the functional unit id to thread id translation array, the threads state, the number of live threads still in the system and the functional units state.

The ready to run container keeps track of the order of threads being created. The semantics of the ready to run container can change from a queue to a stack as desired, it also requires two registers to maintain its bounds. The ready to run container is accompanied by an array that maintains the values of input arguments captured when creating threads. Also, the ready to run container comes with a lookup table that maintains a current translation between functional units and the threads currently running on them. Also, threads state and functional units state are responsible for keeping track of state by marking the corresponding thread or functional unit as busy, free or joinable. If the system is defined as detachable, a counter will be declared to keep track of the number of live threads still active in the system. This register can be read in the main thread of execution to verify if all threads have terminated.

First, threads state, functional units state and joinable state are implemented as bit fields. This allows the library to validate the availability of free threads or functional units instantaneously without the need to loop over an array. State is implemented as registers where each bit represents the state of a single unit or thread. A high bit represents a free resource and a low bit represents a busy resource. To verify the presence of free resources an if statement would do. If there are any free threads or functional units the bit field will evaluate to true. Otherwise, if all were occupied the bit field will evaluate to false. Also, by using bit fields it is easy to retrieve the index of the next free resource by exploiting priority encoders. Efficient and fast encoder macros were provided by the HandelC standard library.

The ready to run container is implemented as a LUT RAM array of unsigned integers. In LUT RAM arrays only one element can be accessed at the same clock cycle to save resources. This access restriction is not harmful. A global semaphore should be locked before accessing this container. That entails only one member of that array is needed at a certain instant in time. Another array that parallels the ready to run container is implemented to hold a copy of the argument object for each created thread till pushed on a functional unit. HCthreads saves a copy of the input argument rather than a pointer because in HandelC pointers are references to locals or globals and should be resolved at compile time. The concept of allocating and deallocating memory in the free store does not stand. If the programmer passes a reference to a local, once the original variable in the calling function goes out of scope the reference to that variable is invalidated. The only option is for the programmer to implement input arguments as globals then passing references of these. Therefore, HCthreads takes care of that uniformly within

the HCthreads implementation. Finally, global registers are included to mark the bounds of the ready to run container and the order of scheduling depending on the desired stack or queue behavior.

One register is used to maintain the number of live threads in the system in case of a detachable scheme. This register is initialized to zero, then incremented inside the thread_create call every time a thread is created and decremented within the termination logic every time a thread finishes. The purpose of this register is for the developer to block on in the main thread of execution after creating the initial thread or threads. In this case the main thread of execution will halt till the number of live threads in the system falls back to zero.

Chapter 6

HCthreads: Supporting Utilities

The HandelC development environment supports various FPGA platforms. Some of these platforms are under the Celoxica brand. The development environment and the build tool chain work seamlessly when targeting these platforms. Also, Celoxica issues support libraries and build scripts for third party platforms. However, Celoxica doesn't offer a global methodology to design and implement HW/SW systems. Limited with the available equipment, and the desire benchmark different C to Hardware translation tools with the same platform, it was decided to develop a solution for this problem. This solution should enable the developers to use HandelC with a wider range of platforms. Xilinx currently holds a large share of the FPGA market. Therefore, it would be rational to think of extending HandelC support to their platforms first. This motivation was also driven by the available ML310 platform.

The ML310 platform is built around the Virtex II Pro FPGA. This FPGA includes, besides the standard reconfigurable fabric, embedded microprocessor cores. For this chip, and others in the Virtex family, hardware co-processors are abstracted in IPIC compatible cores. Using the IPIF these cores can be placed on

either the OPB or the PLB bus. This setup would make the integration between the output of HandelC and the Virtex family possible. Yet, the Hthreads system was a better starting point. Hthreads and the HWTI provide the required high level abstractions for free.

Hthreads is an operating system that abstracts communication and synchronization of hardware co-processors into hardware threads. Hthreads implements an interface called the HWTI on top of the IPIC. The HWTI is a well established interface. It was designed, implemented and thoroughly tested to provide communication and memory abstractions from bus protocols and operations to high level store and load operations. HandelC cores can be arranged nicely into hardware threads using the HWTI. This will achieve the desired integration with Xilinx platforms. Adopting this solution made the HandelC integration effort one of the first solid customers for the Hthreads system.

The first iteration of this integration used the concept of data streaming. The data was streamed via the hardware thread user logic to the HandelC core. The hardware thread user logic then communicates the results back to shared memory. This is detailed in first section in this chapter.

The other scheme replaced the hardware thread user logic completely with HandelC code. That meant the HandelC core will assume an additional responsibility of interfacing with the HWTI. This is presented in the second section.

The fine tuning framework and the underlying assumptions are then discussed in the last section.

6.1 Simple Data Streaming

Hardware threads in the Hthreads system are implemented as sequential state machines under the HWTI. These state machines carry the intended functionality in a fashion very similar to programs implemented at the assembly language level. It would be easy to for the HandelC core to set at one of these states. The hardware thread logic would start execution when it is scheduled to run. Then, the user logic state machine will retrieve the dataset needed by the HandelC core. This data would be transferred to the HandelC core using a couple of simple communication schemes, depending on the implemented algorithm. After that the HandelC core will start working on that dataset. It will signal the user logic state machine when it is done. After that, the results dataset will be transferred back to user logic state machine using the same communication schemes mentioned earlier. Finally, the user logic state machine will communicate the results dataset back to shared memory using the HWTI.

Following the simple description above, one can tell that this scheme benefits from using the user logic state machine by avoiding having to deal directly with the HWTI. The fastest way for developers to write hardware threads in the Hthreads system is to start from a template of the user logic state machine. This template contains the implementation of the state machine according to the HWTI specifications and the description of the minimal set of states needed to code a hardware thread. In order for developers to incorporate a HandelC core in this state machine, they have to add a bunch of additional states for data handling and synchronization. This scheme is straight forward and quick to implement. Unfortunately this scheme comes with two major drawbacks, the first being the additional clock cycles needed for this additional synchronization and

the other being cutting off the HandelC core from the abstractions the HWTI and the Hthreads system provide.

The number of cycles required for memory operations in the Hthreads system depends on bus traffic and on the location of the memory segment being accessed. Therefore, the communication framework between the user logic state machine and the HandelC core should include synchronization signalS to flag when either side of the channel is ready to read or write data. An additional overhead will be added to the latency of memory operations. Also, this communication scheme cannot be unified over different algorithms. Certain algorithms might require a large dataset at a time such as QuickSort and Laplace while others might require just one word as input such as FIR and NQueens. Hence, this communication scheme has to be implemented manually and reevaluated on problem to problem basis making it unattractive.

The second drawback of this scheme would be the obscuring of the HandelC core from the abstractions offered by the HWTI. This is a major concern. To prove this point it is sufficient to only mention one scenario where the HandelC core fails to carry out its operation. Assume the HandelC core is supposed to sort a dataset of a large number of records, the streaming scheme would have to transfer the entire dataset or at least a considerable portion of it into the HandelC core first, and this is quite clearly infeasible. Instead, it would be better if the HandelC core had access to the off ship DRAM where such huge dataset resides. For this reason the data streaming scheme fails to address all communication needs.

To solve the problems mentioned above a new method should be used. The hardware thread user logic state machine should be bypassed. The HandelC core should be able to interact to the HWTI directly.

6.2 Full Integration with Hthreads

HandelC is cycle accurate, which means it behaves like an HDL. That makes it possible for HandelC to synchronize correctly with the HWTI. This is done by unifying the interface the HandelC core exposes to the outside world to become the same interface the user logic state machine entity used to expose. On the other end, this interface will be wrapped by shared functions within the HandelC core. Every time the user requires an action from the HWTI a simple call to one of these functions would do. For instance, should the user within the HandelC core require dereferencing a pointer within the shared memory address space, a call to the load function passing the address as a parameter will do. Then the HandelC core will issue the proper opcode to start a load operation. At the same time the load function would take care of the timing requirements imposed by the HWTI. It will drive the ports for the required number of clock cycles, and then wait for the proper synchronization signal from the HWTI and ultimately deliver the result back to the user in a form of a return value. Likewise, other operations such as pop, push, thread exit and save are wrapped within similar functions. Just like in the data streaming scheme the user does not have to deal with the HWTI directly, nonetheless the user can request all HWTI services with very little effort.

Currently, this method is done by including the Hthreads support library hthreads.hch when developing a new hardware thread using HandelC. The library currently contains implementations for the basic memory and synchronization operations such as load, store, push, pop, thread exit and memory allocation operations. Adding other opcodes supported by the HWTI such as Hthreads mutex operations would be simply to add new shared functions to the library for each

new opcode. Coding these functions is a simple matter. The parameter list for a given function should reflect what the HWTI expects the user to provide for that opcode. The synchronization code required for all opcodes is the same.

6.3 HCthreads Fine Tuning

This framework was devised to help HandelC developers assess the required recourses for the HCthreads library for different applications. It would be naive to assume that a given HandelC core will utilize every functional unit just because these components are there. On the other hand, it might be the case where more functional units or bigger status containers are needed to achieve more parallelism. Being able to assess what the algorithm under study actually needs in terms of these components might yield some modest savings in space or it might provide a boost in performance.

HandelC is cycle accurate, thus simulating a HandelC implementation for a given input would yield the exact same result of running that implementation in hardware. This allows developers to debug their HandelC code without having to convert it first to a netlist or an HDL format and then simulate using third party tools. Also, developers using the fine tuning capability of the HCthreads library can assess if their implementation might require less or additional HCthreads resources. This is achieved by defining the DEBUG compilation flag, running the application and then investigating certain globals that indicate the resource usage of the HCthreads components.

The only concern developers should take care of is the interaction of their cores with the HWTI. Memory access times in the HWTI are dependent on the portion of the shared address space to be accessed [2]. To solve this problem

the framework can either assume the problem away by letting all access times to take one clock cycle or by modeling memory latencies. Modeling these memory latencies should be relatively easy. First, the framework can safely model the access times as a constant for the BRAM within the hardware thread the HandelC core resides in. The probability of another thread accessing this portion of the shared address space is almost NULL. On the other hand, access times for the off chip RAM can be modeled by a different constant. This approximation should be close enough to the actual case with an acceptable margin of error. This can be rationalized easily by noting that the general case of HandelC cores under the Hthreads system would consist of only one hardware thread in the entire system. This hardware thread wraps around the HandelC core to provide FPGA platform support. There the main thread of execution is blocked on the CPU trying to join on that single hardware thread once it creates it. Therefore, there is no need to expect heavy bus contention coming from other hardware or software threads. Finally, programmers can readjust their HCthreads library according to the initial profiling findings and repeat the process to their satisfaction. Being able to carry out this iterative operation in the simulator rather than the real system will save time.

Chapter 7

Results

The first objective of these tests is to demonstrate the programming model enhancement when expressing TLP using HCthreads. The second objective is to show that this enhancement does not come with severe performance costs compared to the ILP technique in HandelC. The two programming models are presented in the first section of this chapter.

The second and third sections deal with simple test cases implemented on two platforms. The first platform is the HandelC simulator meant to present the ideal case. The simulator abstracts memory latencies because the global arrays and other data structures in the system require only once clock cycle to be accessed. Speedups or slowdowns in execution times would be clearly visible in this platform. The other targeted platform in these tests is the Xilinx ML310. This platform is supposed to display the capabilities of HCthreads in a market available platform with memory access limitations and other practical HW/SW considerations.

The tests executed on these platforms are simple algorithms with different threading and memory characteristics. These tests are implemented in two forms. The first form is the parallel implementation of these algorithms using the na-

tive HandelC construct par. Par acts as a simple scheduler and terminator for parallel copies of the accelerated function. This form does not require extra coding except when it comes to recursive algorithms. In recursive algorithms a stack should be implemented to support recursion where HandelC does not. The second implementation of these test cases is done using HCthreads.

The first algorithm is Quicksort which is a recursive memory intensive program suitable for a ready to run stack and a detached threading behavior. The second algorithm is the solution to the NQueens problem. This is an embarrassingly parallel algorithm with limited memory interaction suitable for a ready to run stack arrangement under a detachable scheme. The final algorithm is a simple implementation of a Matrix Multiplier. This program is suitable for a ready to run queue under either a joinable or a detachable solution. Investigating the HCthreads library with these algorithms covers testing for the behavior of the ready to run container and the employed joinable or detachable threading scheme.

Finally, additional test cases are to verify the supporting utilities. The correctness of the Hthreads support library is discussed in the fourth section. Then in the fifth section, some tests would shed a light on the validity of the proposed fine tuning framework.

7.1 Programming Model Enhancement

The case to be made in this experiment is the facility HCthreads provides in implementing TLP over other ILP implementations in HandelC. Though quantitative results cannot be presented, qualitatively it is reasonable to state that a parallel implementation using HCthreads is easier to code. Programmers these days are familiar with the Pthreads interface. With few pointers and hints these

programmers can express parallel algorithms easily in HandelC using HCthreads. The initial investment is minimal. On the other hand, it will be inconvenient for programmers to implement parallel and recursive algorithms with just ILP constructs. There is no predefined method, programmers need to come up with their own parallelization framework. They need to code and test the correctness of these implementations themselves. Consequently, the initial investment is not marginal.

As shown in Figure 7.1 HCthreads calls are similar to Pthreads calls in terms of syntax and semantics. The major difference rests in the extraction of the properties of each individual thread to global definitions enforced over all threads spawned in that particular solution. The other difference in syntax lies in the signature of the hcthread_create and the hcthread_join functions where the language pointer semantics and limitations forced the interface to be defined in this fashion. Refer back to Section 5.3.

```
structAddr = pop();
size   = load(structAddr);

/* initialize input */
arg.startIndex = 0;
arg.endIndex   = size[16:0] - 1;

/* create initial thread */
hcthread_create(arg);

/* block till all terminate */
while(LiveThreads) delay;

push(ticks);
threadexit();
```

```
while(row < DIMENSION1)
{
    Flag = hcthread_create(&thread[Top], row);

    if(!Flag)
    {
        hcthread_join(thread[End]);
        End++;
    }
    else
    {
        row++;
        Top++;
    }
}
```

Figure 7.1. Creating and Joining on Threads using HCthreads

On the other hand, as portrayed in Figure 7.2, parallelism using pure ILP HandelC constructs is not straight forward. While in the case of simple iterative algorithms such as matrix multiplication, implementing parallelization is rather simple. The programmers need only to worry about creating multiple copies of the accelerated function and to coordinate the concurrent invocation of these

functions using the par construct. However, in the case of recursive algorithms programmers have to add recursion support themselves. Even though the average programmer will be able perform such a task effectively, HCthreads already offers recursion support for free.

```
/* parameters */
#define NO_FUNITS  5
#define STACK_SIZE 14
#define ARRAY_SIZE 240

/* type definitions */
typedef unsigned int 8 UINTI; // Data Array Index
typedef unsigned int 4 UINTS; // Stack Index
typedef unsigned int 1 bool;

/* stack structure */
typedef struct
{
    UINTI startIndex;
    UINTI endIndex;
} StackData;

sema  StackGaurd;
UINTS StackTop;
ram   StackData Stack[STACK_SIZE];

void QuickSort[NO_FUNITS]();

/* main thread of execution */
void main (void)
{
    /* initialize stack */
    StackTop = 1;
    Stack[0].startIndex = 0;
    Stack[0].endIndex   = ARRAY_SIZE - 1;

    /* call all functions in parallel */
    while(StackTop)
    {
        par(i = 0; i < NO_FUNITS; i++)
        {
            QuickSort[i]();
        }
    }
}
```

```
void QuickSort[NO_FUNITS]()
{
    bool terminate;

    while(trysema(StackGaurd) == 0) delay;
    if(StackTop)
    {
        StackTop--;
        terminate = 0;

        startIndex = Stack[StackTop].startIndex;
        endIndex   = Stack[StackTop].endIndex;
    }
    else
        terminate = 1;
    releasesema(StackGaurd);

    if(terminate)
        return;

    /* actual algorithm goes here */

    while(trysema(StackGaurd) == 0) delay;
    if( leftIndex != startIndex )
    {
        Stack[StackTop].startIndex= startIndex;
        Stack[StackTop].endIndex = leftIndex;
        StackTop++;
    }

    if( rightIndex != endIndex )
    {
        Stack[StackTop].startIndex = rightIndex;
        Stack[StackTop].endIndex = endIndex;
        StackTop++;
    }

    delay;
    releasesema(StackGaurd);
}
```

Figure 7.2. Recursion Support & Function Parallelization using ILP Constructs

7.2 Simulator Results

The experiments carried out on this platform proved HCthreads quite useful. As illustrated in the associated figures and tables HCthreads performed better than the ILP implementation almost in all cases. First, for the Quicksort algorithm the objective was to sort one thousand integers. The presented data shows that the HCthreads implementation causes an additional overhead when compar-

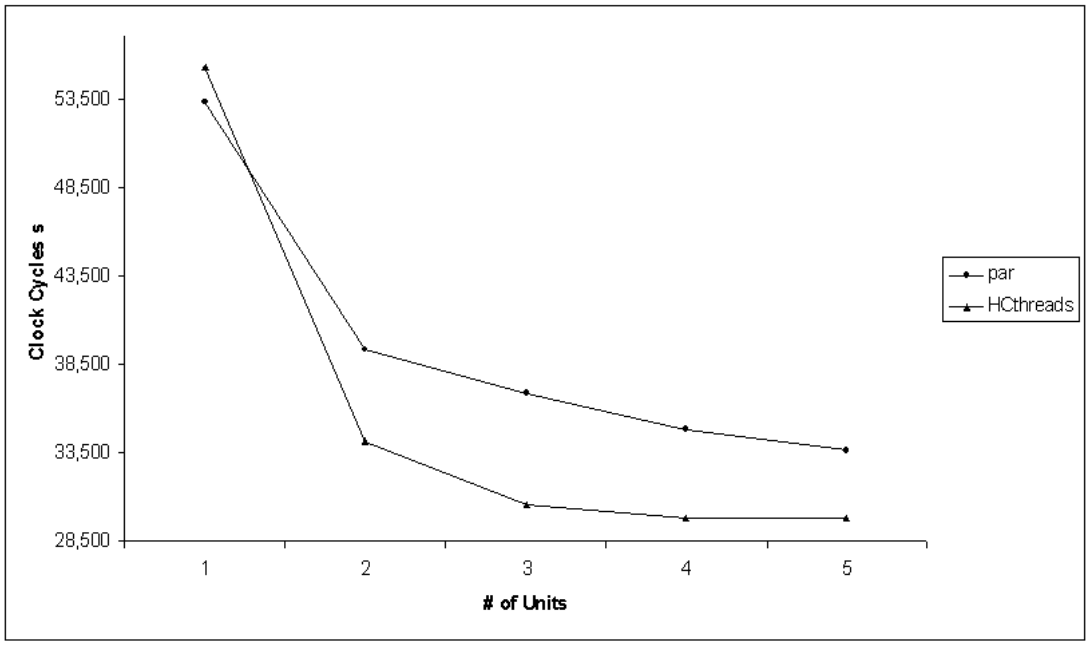

Figure 7.3. QuickSort Total Clock Cycles, Simulator

ing single units. However, when the number of functional units is increased the drops in total clock cycles will lessen the effect of the scheduling and termination overhead. The reason behind this drop is that HCthreads schedules and terminates tasks individually while the par construct does that in groups. In the case of HCthreads a thread will be assigned to a functional unit right when created and the functional unit will be freed when the current thread completes its computation. On the other hand, the par construct will invoke multiple copies of the accelerated function at the same instant, and it will wait for all these functions to terminate before the second invocation round can commence. This means HCthreads hardware utility is higher than the par solution. Also, HCthreads reached its saturation point at least two units before the ILP implementation. In actual systems this means savings in allotted physical resources and better timing results.

The results for the Eight Queens algorithm were also promising. Both parallel

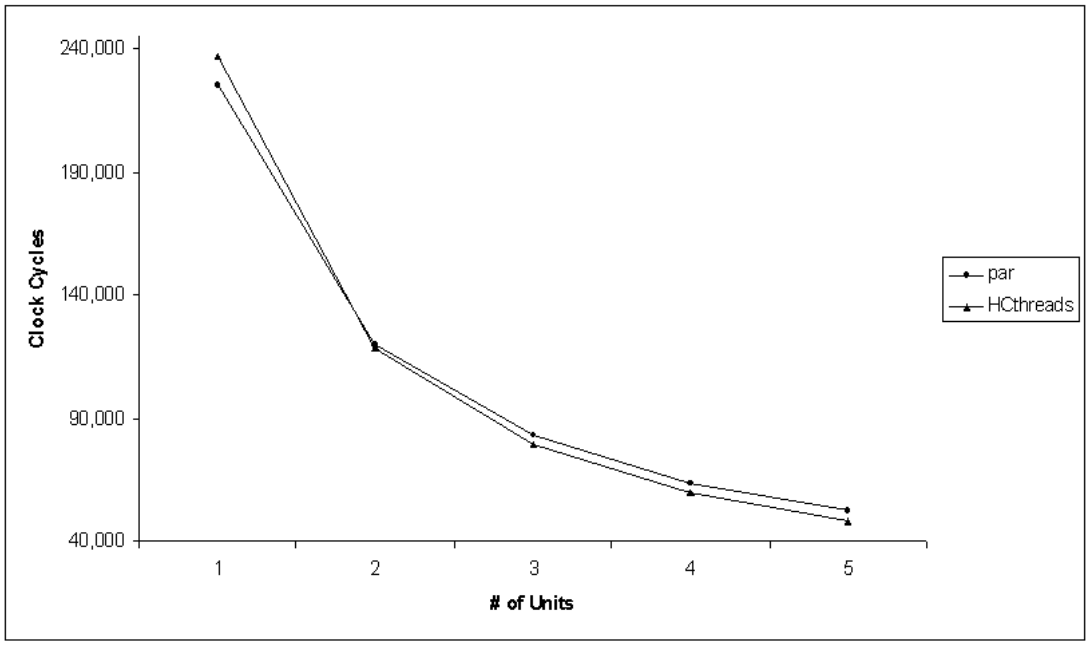

Figure 7.4. NQueens Total Clock Cycles, Simulator

implementations were on an equal footing. Tasks in the NQueens solution take almost the same number of clock cycles to terminate. That means there is no big difference between scheduling threads in groups or independently. However, the HCthreads implementation is faster by almost four thousand cycles when having three functional units or more.

The Matrix Multiplier algorithm was to multiply two 14X14 matrices. Both parallel implementations showed speedups over the single unit. Still, HCthreads performing better in almost all instances. However, in the case of three functional units both parallel implementations achieve their best results. The ILP implementation starts to slowdown after that. The reason behind such behavior can be understood by explaining a reported bug in semaphore operations in HandelC. For this algorithm the accelerated function is implemented using two nested for loops with each taking exactly three clock cycles to execute; only one of these clock

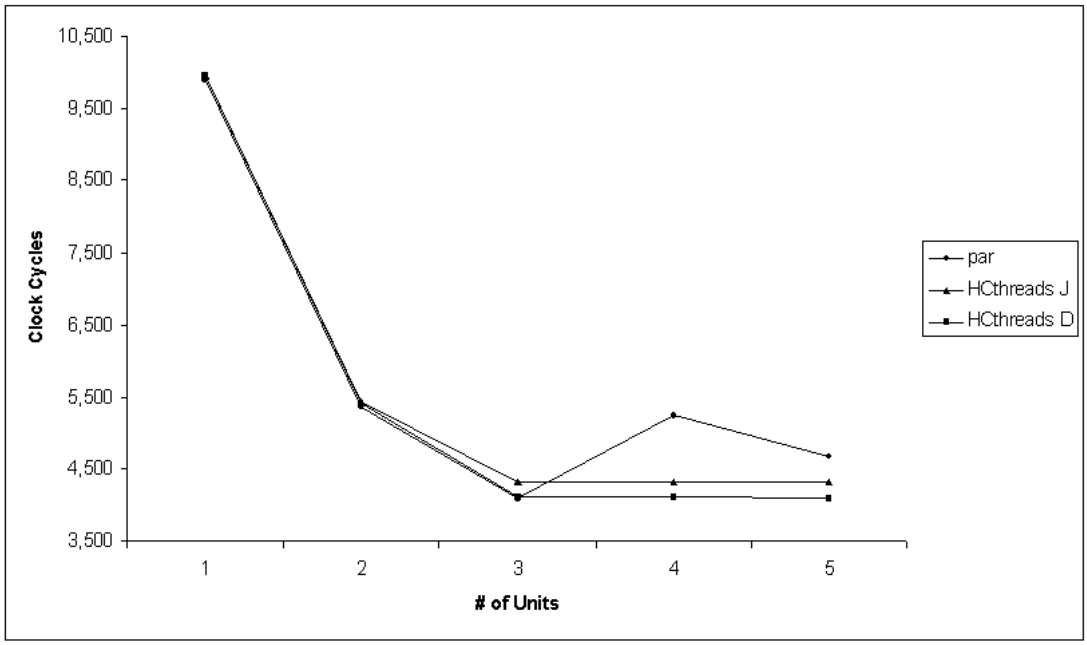

Figure 7.5. Matrix Multiplier Total Clock Cycles, Simulator

cycles in both loops is protected with a semaphore. When running the algorithm with three functional units, there would be no contention on the any semaphore after the first inner iteration. However when running the algorithm with four functional units there would be two functional units competing on the inner most semaphore at any clock cycle. The assumption is that once the third functional unit releases the semaphore the fourth functional unit should acquire it, that is the case. Then the fourth unit is waiting already for a couple of clock cycles but the first unit is about to join that waiting queue. At the next clock cycle right when the semaphore is released control will be given to the first unit because it is listed first in the code. The clock cycle after that, the second and fourth units will be competing for that semaphore, again control will be given to the second, then the third then the first again. The fourth unit will never get that semaphore till the first three exit these loops. Thus, running four units in parallel will be

similar to running three units in parallel and after that one unit in sequence. That will force the total number clock cycles to almost double when scheduling a single batch, even though the expected behavior would be for the total number of clock cycles to saturate when adding more unit after three. However, this bug in semaphore arbitration is not an issue with HCthreads. Unlike in the ILP implementation where threads are scheduled and terminated in groups, threads in HCthreads are independently scheduled right when created and terminated right when done. If one functional unit gets stuck the first three will work till all threads are done, then the fourth unit will free itself and finish its assigned computation. Finally, the detachable HCthreads implementation required less clock cycles to run compared to the joinable build. Thus the recommendation of implementing with detachable rather than joinable is valid.

Table 7.1. Simulator Total Clock Cycles

	par construct				
	1	2	3	4	5
QS	53,338	39,370	36,901	34,833	33,594
NQ	225,063	119,712	82,810	63,557	52,099
MM	9,888	5,373	4,083	5,241	4,676
	HCthreads				
	1	2	3	4	5
QS	55,335	34,124	30,523	29,746	29,761
NQ	236,791	118,627	79,383	59,662	47,926
MMJ[1]	9,961	5,423	4,313	4,313	4,313
MMD[2]	9,947	5,402	4,107	4,107	4,084

[1]Joinable Design
[2]Detached Design

7.3 ML310 Results

This section captures the implications of HCthreads when targeting a real FPGA platform. In this platform, the HandelC component will reside as user logic inside a hardware thread. This hardware thread using the HWTI will bridge the gap between the HandelC component and the rest of the system. To capture most accurately the total number of clock cycles passed during a single run. The HandelC component incorporates a counter that starts counting clock cycles when the component receives a thread start signal and resets itself when the thread exits. The processor can schedule the hardware thread to run for an arbitrary number of times, in our case a hundred. Every run the processor aggregates the total number of clock cycles. Eventually the processor will average that aggregate and report that number back as the total number of clock cycles needed for a single run. This way the unknown effects of bus contention when accessing memory will be distributed over a large set of samples.

First, the Quicksort algorithm will sort a set of ten thousand thirty two bit integers. Looking at the results in Figure 7.6 the total execution time will get worse as the number of units increases. The reason behind such slowdown is that the memory latency when accessing the off chip DRAM is relatively large and Quicksort is a memory intensive application. That entails the portion of the code that can be accelerated by the presence of multiple units spends far less time executing than the synchronized portion responsible for memory interaction, this is a classic case of the limitation of Amdahl's Law. On the other hand, when more units are added to the system more reconfigurable resources are needed generating more competition over routing resources. Coupled with that the circuit for the semaphore acting as a memory guard will get more complex, thus the

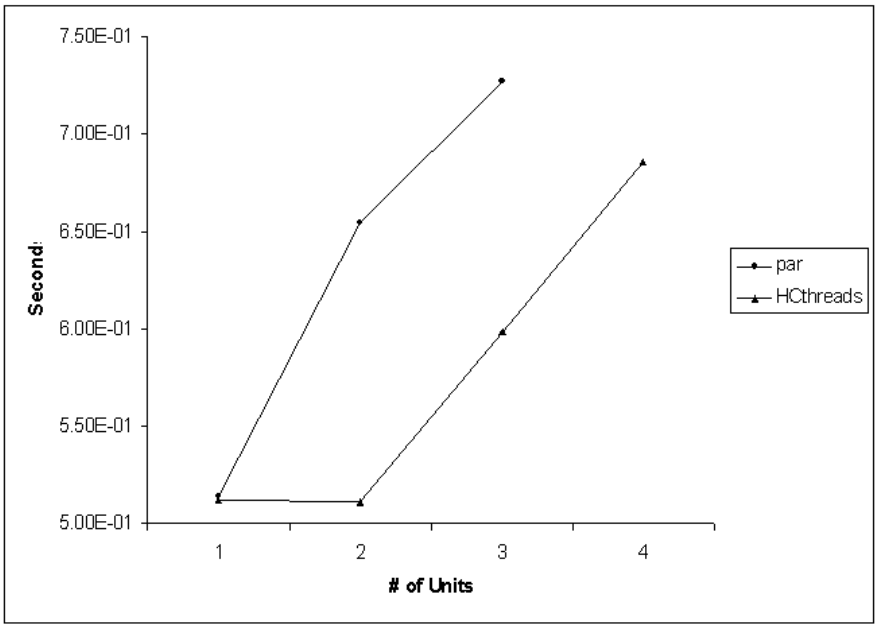

Figure 7.6. QuickSort Total Execution Time, ML310

maximum delay on the longest combinational path will get worse. Both these factors contribute to worse clock speeds with every added unit forcing the total execution time to increase, even though the total number of clock cycles remained virtually constant. In the case of Quicksort all gains achieved in the form of total clock cycle reductions is exceeded by the longer clock durations. Although these are negative results, there are a couple of positive points to report on. The first point is the slight overhead of HCthreads when having only one unit running in the system. That indicates the HCthreads library is an acceptable solution if programmers are only worried about adding recursion support to their HandelC implementations. The other positive result would be the slower rate in which clock speeds fall when using HCthreads over the other. That indicates HCthreads builds combinational circuits with smaller stage delays for the same number of units. This was evident when the ILP implementation using four units even failed

to run, while HCthreads implementation using four units provided better results than the ILP build with three units.

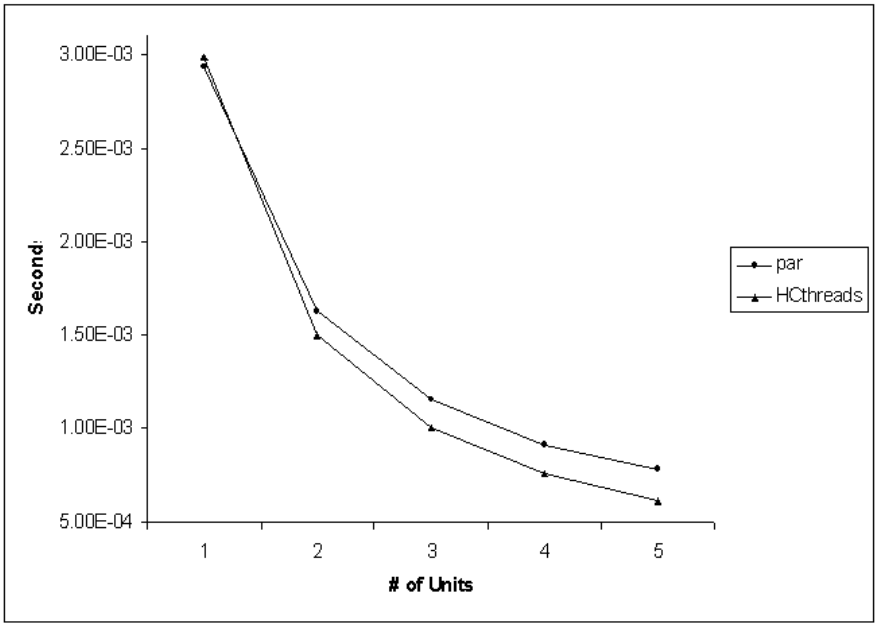

Figure 7.7. NQueens Total Execution Time, ML310

The Second test was the solution for the NQueens algorithm with eight queens. The results for this algorithm are quite promising. HCthreads didn't just achieve modest speedups over the single unit case, but managed to achieve speedups over the ILP implementation given the same number of units.

The Matrix Multiplier was supposed to multiply a 25X9 times a 9X13 matrix. This algorithm achieved slowdowns for the same reasons Quicksort did, too little to accelerate and worse clock speeds. However, just as it was the case for Quicksort, HCthreads implementations provided better timing results compared to their ILP counterparts.

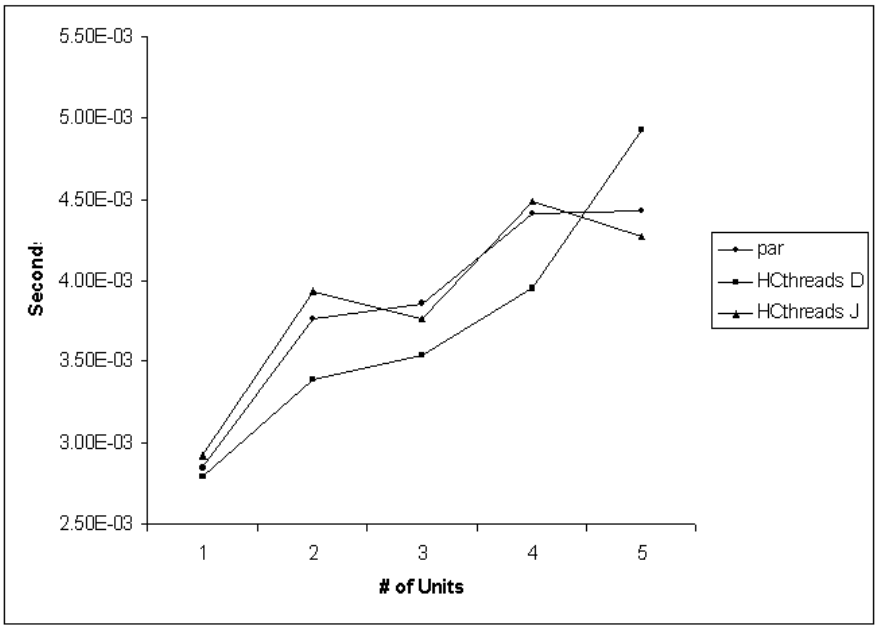

Figure 7.8. Matrix Multiplier Total Execution Time, ML310

Table 7.2. ML310 Total Clock Cycles

	par construct				
	1	2	3	4	5
QS	41,079,840	41,149,963	41,142,549	NA [1]	NA [2]
NQ	236,422	131,187	93,018	73,814	62,844
MM	224,534	221,606	225,799	221,805	215,298
	HCthreads				
	1	2	3	4	5
QS	41,072,433	41,061,839	NA [3]	41,055,250	NA[1]
NQ	238,324	120,794	80,906	60,999	49,115
MMD[4]	225,035	213,706	213,404	213,400	213,397
MMJ[5]	224,690	213,561	213,529	213,507	213,821

Table 7.3. ML310 System Minimum Period in nSeconds

	par construct				
	1	2	3	4	5
QS	12.52	15.91	17.68	18.35	NA
NQ	12.42	12.41	12.49	12.42	12.44
MM	12.67	17.01	17.10	19.91	20.56
	HCthreads				
	1	2	3	4	5
QS	12.46	12.45	NA	16.71	19.71
NQ	12.54	12.42	12.41	12.51	12.42
MMD	12.40	15.86	16.57	18.52	23.07
MMJ	13.02	18.43	17.63	21.02	19.99

Table 7.4. ML310 Slices

	par construct				
	1	2	3	4	5
QS	2989	3641	4216	4872	NA
NQ	2541	2755	2914	3120	3253
MM	2612	3011	3373	3871	4230
	HCthreads				
	1	2	3	4	5
QS	3387	3958	NA	5342	6114
NQ	3771	4017	4333	4651	4882
MMD	2737	3166	3535	4013	4454
MMJ	3021	3405	3789	4299	4691

7.4 Testing HandelC and Hthreads Integration

No separate test cases were carried out for this portion, yet the support library was deemed satisfactory. When testing on the ML310 platform all HandelC cores were placed inside a hardware thread. Therefore, the Hthreads support library

[1]System froze when executed
[2]System was not built, no need to generate results when the previouse case failed
[3]System could not be synthesized due to an error in the Xilinx tool chain
[4]Joinable Design
[5]Detached Design

Table 7.5. ML310 Core Minimum Period in nSeconds

	par construct				
	1	2	3	4	5
QS	9.90	12.44	13.71	13.86	NA
NQ	9.62	9.64	9.84	9.86	10.00
MM	10.83	13.12	12.91	14.37	14.92
	HCthreads				
	1	2	3	4	5
QS	9.29	10.54	NA	13.93	14.90
NQ	8.83	9.19	9.32	9.05	9.15
MMD	10.27	12.52	13.44	13.89	14.98
MMJ	10.27	12.91	13.36	14.45	15.03

was present in about thirty different test cases with millions of back and forth transactions. These transactions contained the following opcodes and functions:

- OPCODE_LOAD
- OPCODE_STORE
- OPCODE_PUSH
- OPCODE_POP
- OPCODE_CALL
- U_FUNCTION_RESET
- U_FUNCTION_USER_SELECT
- U_FUNCTION_START
- FUNCTION_HTHREAD_EXIT
- FUNCTION_MALLOC
- FUNCTION_FREE

All these functions and opcodes performed as expected with no exception. However, because one hardware thread were running there was no need to use Hthreads mutexes, conditionals and other. To verify if this set of functions work properly with the support library, a couple of pilot programs were constructed and tested with successful results, with the exception of one scenario dealing with mutexes still under investigation. The support library should undergo complete verification to make sure that differences in the programming model between HandelC cores and the user logic state machine it replaced is not severe enough to corrupt future complex implementations.

7.5 Fine Tuning Results

The underlying assumption behind the fine tuning framework is that the simulator would be able to replicate the operation of a real system with random memory latencies to an acceptable degree. If this assumption doesn't hold, profiling should be carried on the system itself to correctly identify the minimum required resources for that problem. But if the assumption holds profiling can be done on the simulator shortening the development cycle.

For this set of tests, the discrepancy will be recorded between the total number of clock cycles produced for a certain algorithm using the simulator with artificial constant memory latencies on the one hand and using the ML310 on the other. Unfortunately, the results were negative. As seen in Table 7.6, the differences in the case of Quicksort are outside any acceptable range, thus rending the assumption invalid. In the case of memory intensive applications any profiling should be carried out on the real system and not in the simulator.

[2]System froze when executed

[1]System could not be synthesized due to an error in the Xilinx tool chain

Table 7.6. Total Clock Cycles for Actual and Approximated Systems

	NQueens				
	1	2	3	4	5
Actual	238,324	120,794	80,906	60,999	49,115
Aprox.	241,294	121,171	81,217	61,330	49,354
Diff.	2,970	377	311	331	239
Ratio to Actual	1.25%	0.31%	0.38%	0.54%	0.49%
	Quicksort				
	1	2	3	4	5
Actual	1,326,236	1,313,656	NA [1]	1,312,301	NA[2]
Aprox.	1,167,646	1,138,934	1,138,793	1,138,872	1,138,831
Diff.	158,590	174,722	NA [1]	173,429	NA[2]
Ratio to Actual	11.96%	13.30%	NA [1]	13.22%	NA[2]

Chapter 8

Conclusion and Future Work

The results proved HCthreads worthy, it served its initial purpose by making TLP easier to express in HandelC with the added speedup over conventional parallelization schemes. Also, the extra area and timing overhead caused by HCthreads can be tolerated in the case of massively parallel applications. The results reiterated the fact that TLP cannot be achieved without breaking the memory bottle neck first and reaching close to true data parallelism [18]. Channels and message passing might be one way to tackle this problem. Unfortunately, the HCthreads system currently cannot show relevant results, because Hthreads does not support such programming models yet.

With respect to future work, the Hthreads support library should be rigorously tested. This should be done by recoding the original test cases for the HWTI into HandelC. The results should be cross examined with previous tests. It would be expected to encounter some variations in these results. The reason behind such difference is the distinction in the programming models between the two implementations, primarily the lack of state in HandelC cores. This difference would not render the Hthreads support library useless, it would still be beneficial

in most cases. Nevertheless, points of weakness should be identified and avoided, after all FPGAs are supposed to be flexible.

Second, The Hthreads support library and the HandelC language can be enhanced by incorporating the globally distributed memory address space directly into the HandelC language address space. Currently, global memory accesses are done via function calls that wrap synchronization logic with the HWTI. Hiding these functions behind standard pointer operators will push HandelC closer to becoming the language for HW/SW co-design.

Finally, additional room can be exploited in the fine tuning framework. The idea would be to use a random variable with a certain distribution to model memory latencies during simulation instead of constants. If successful in this endeavor, profiling can be pushed back to the simulator. This will cut down on the development cycle which will make the HCthreads system more desirable.

References

[1] J. M. Agron. Run-Time Scheduling Support for Hybrid CPU/FPGA SoCs, 2005.

[2] E. Anderson. Abstracting the Hardware / Software Boundary through a Standard System Support Layer and Architecture. 2007.

[3] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. 2006.

[4] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hThreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Sicily, September 2005.

[5] P. Athanas and H. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis, 1993.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] T. Callahan. Automatic Compilation of C for Hybrid Reconfigurable Architectures.

[8] T. Callahan, J. R. Hauser, and J. Wawrzynek. The GARP Architecture and C Compiler, 2000.

[9] Celoxica. Handel-C Language Reference Manual, 2005.

[10] P. Diniz, M. Hall, and J. Park. Bridging the Gap between Compilation and Synthesis in the DEFACTO System. 2001.

[11] S. A. Edwards. The Challenges of Synthesizing Hardware from C-Like LanguagesSeamless Hardware-Software Integration in Reconfigurable Computing Systems. *IEEE Design And Test of Computers*, 2006.

[12] G. Estrin. Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. 2002.

[13] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized Generation of Data-Path from C Codes. In Proceedings of the ACM/IEEE Design Automation and Test. 2005.

[14] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformation. 2003.

[15] KU Hybridthreads. Project Wiki. http://wiki.ittc.ku.edu/hybridthread/Main_Page.

[16] E. Lübbers and M. Platzner. ReconOS: An RTOS supporting Hard- and Software Threads. In *17th International Conference on Field Programmable Logic and Applications (FPL)*, Amsterdam, Netherlands, August 2007.

[17] O. Mencer. ASC: A Stream Compiler for Computing with FPGAs. 2005.

[18] A. Nakajima and R. Kobayashi. Limits of Thread-Level Parallelism in Non-Numerical Programs. 2006.

[19] I. Page. Constructing Hardware-Software Systems from a Single Description. 1994.

[20] S. Russel and P. Norvig. *Artificial Intelligence, A modern Approach*. Prentice Hall, 2003.

[21] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 6th Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[22] G. Stitt and F. Vahid. Thread Warping: A Framework for Dynamic Synthesis of Thread Accelerators. In *Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, 2007.

[23] T. Ungerer. Multithreaded Processors. In *British Computer Society*, 2002.

[24] M. Vuletic, L. Pozzi, and P. Ienne. Seamless Hardware-Software Integration in Reconfigurable Computing Systems. *IEEE Design And Test of Computers*, 2005.

[25] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin, and C. Kitchen. An overview of FPGAs and FPGA programming; Initial experiences at Daresbury. 2006.

[26] D. W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS*, pages 176–188, 1991.

[27] M. J. Wirthlin and B. L. Hutchings. DISC: The Dynamic Instruction Set Computer, 1995.