

HTHREADS: A COMPUTATIONAL MODEL FOR RECONFIGURABLE DEVICES

Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, David Andrews

Information and Telecommunication Technology Center - University of Kansas
2335 Irving Hill Road, Lawrence, KS
{peckw,eanderso,jagron,jstevens,bricefab,dandrews}@ittc.ku.edu

ABSTRACT

Recent architectural advancements in reconfigurable devices have exposed the ability to support massive parallelism inside of small, low-cost, embedded devices. The massive parallelism inside of these reconfigurable devices has promised to bring an unprecedented level of performance to the embedded systems domain. However, the complexity of programming these reconfigurable devices is daunting—too daunting for the average programmer. This paper presents hthreads. Hthreads is a computational architecture which aims to bridge the gap between regular programmers and powerful but complex reconfigurable devices. Hthreads accomplishes this goal using three layers of abstraction built upon standard reconfigurable devices: an operating system capable of supporting a diverse collection of computational models within a reconfigurable device, an intermediate form representation which eases the development of applications on reconfigurable devices, and support for high level languages which are familiar to most programmers.

1. INTRODUCTION

The recent advances in reconfigurable devices have introduced massive computational capabilities into the embedded systems domain. Additionally, Moore's Law promises that the computational capabilities will increase an order of magnitude over the next few years. However, programming support for these devices is failing to keep pace. Hthreads is a computational model which aims to align the programming capabilities of reconfigurable devices to the computational capabilities of those devices.

The hthreads computational model is a set of cooperating layers of abstraction which form a bridge from high level programming languages to low level reconfigurable devices. The first of these layers is the operating system layer which provides run-time support to a diverse set of computations. The intermediate form layer is built on top of this and provides a common format for describing computations which run on reconfigurable devices. The last layer is the high level language support layer which builds upon the intermediate

form to support languages which are easy to use and familiar to most programmers.

2. OPERATING SYSTEM

The first layer in hthreads is the operating system layer. This layer provides run-time support to a diverse collection of computational models which run on reconfigurable devices. This capability is achieved by defining a standard interface to which computations must conform in order to participate in the system. We have chosen to use the thread interface.

The thread interface is a standard interface which supports monitoring and controlling the execution of computations in a system. This is done by requiring that critical tasks be performed by the operating system on behalf of a computation's request. Using the information from the request, the operating system can then make global decisions about how to correctly control the execution of a system. In addition, computations are required to carry out any commands that the operating system might issue in response to a request. For example, suppose a computation in hthreads needs to access some shared memory. To do this the computation must first request a semaphore lock from the operating system. If the semaphore is already locked then the operating system will inform the requesting computation that it must halt its execution. In this case the requesting computation is required to halt its own execution. It is important to note that the operating system does not halt the computation but instead tells the computation to halt itself. This allows hthreads to service requests from computations without needing to know the underlying computational model used by the computation.

2.1. Design

The thread interface used by hthreads is based around three major tasks: management, scheduling, and synchronization. Each of these tasks is designed around a simple request protocol. The protocol involves three steps. First, a computation will send a request to the operating system. Second, the operating system will service the request and return status

information back. Third, the requester will process the returned information and must perform any commands issued by the operating system.

The first step is performed by issuing a request in the form of a standard memory read to an address in the operating system's address range. Any information about the request is encoded into the address being accessed. A more detailed description of the request protocol is given in [1]. The second step is then performed by a state machine which is part of the operating system. This state machine will calculate the correct response to the request and return that response as the value of the memory location which was read. The requester will then take the value returned from the memory read and interpret it according to the semantics of the requested operation. It is up to each individual thread to interpret the returned value and take the appropriate action.

2.2. Implementation

To implement the three major tasks of the hthreads operating system (management, scheduling, and synchronization) we have developed three hardware based state machines using VHDL. Because each of these state machines are separate hardware components each can execute in true parallel with the others and with the application itself. This provides coarse-grained parallelism which is needed for high performance.

The implementation of all of the hardware based service providing state machines in the operating system is based around providing a set of memory mapped virtual registers. These registers serve as a mechanism by which information is passed to a state machine through an address during a memory read operation. We say these registers are virtual because they have no physical representation in hardware. The use of virtual registers has allowed the hthreads operating system to remain simple and efficient while supporting many different computational models.

The first state machine that we developed using the virtual register design was the thread management component of the operating system. This component maintains global resources, such as thread identifiers and state information, in the operating system. The thread management component works in cooperation with the thread scheduling component to provide most of the functionality in the operating system. The scheduling component performs all of the scheduling computations for the operating system. This includes managing a ready-to-run queue and maintaining scheduling state information for all computations running in the system. The last component developed was the synchronization component which manages all of the synchronization primitives in the operating system. This includes tracking semaphore status information and maintaining waiting queues. One interesting feature of the synchronization primitives in our oper-

ating system is that they are based on the virtual register design. This is an interesting departure from more traditional designs which are based on carefully crafted sequences of instructions inside of a particular instruction set architecture. These designs require caches and complex coherency protocols to maintain efficiency. Our design frees itself from a fixed instruction set and performs efficiently without the need for caches or complex coherency protocols. More detailed information about the implementation of the hthreads operating system along with its performance characteristic can be found in [1, 2]

2.3. Hardware Thread Interface

To ease the use of operating system services in application hardware we have developed a simple hardware API known as the hardware thread interface (HWTI). The HWTI is designed as a hardware resident finite state machine just like all of the other state machines in the operating system. Unlike the other state machines, however, the HWTI exists as part of the application hardware on a per thread basis. The HWTI hides the complexities of communicating correctly with the service providing components in the hthreads operating system. It is capable of correctly forming a service request for any of the services provided by the operating system and can correctly interpret return values from the requests.

The HWTI is designed as a finite state machine with two interfaces. The first interface is the system interface which maintains five memory-mapped registers. These registers represent the context of a computation running in the system. The identifier register is the first register and it is filled in by the operating system when the computation is spawned during execution. The next register is the status register and it is a read-only register which contains information about the computation's state. Its primary purpose is for debugging. The command register is the third register and is used to inform the computation about some action that it must take. Typically this register is used to wake the computation after it has been suspended. The fourth register, the argument register, is used to pass an argument to the computation when it is spawned. Typically this value will be a pointer to some memory location which contains more information. The last register is the result register which is used to pass a result value from the computation back to the system. This information can then be given to another computation if needed.

The second interface is the computation interface and it is also composed of five registers. These registers abstract away the complexities of interacting with the operating system and are only available to the computation (they are not memory mapped). The first register in the computation interface is the status register which is used to inform the computation about its own status from the operating system's perspective. This includes information such as whether the

computation should be executing, whether the computation should reset itself, or whether the HWTI is performing a service request on behalf of the computation. The next register is the opcode register; it is used by the computation to indicate what service the HWTI should perform. For example, if the computation needs to lock a semaphore then it would need to set the opcode register to LOCK. Argument 1 and argument 2 are the third and fourth registers in the interface and they are used to provide additional information to the HWTI. For example, the HWTI must know what semaphore to lock before it can perform the operation. Thus, the computation must load argument 1 with the number of the semaphore to lock before loading the opcode register. The last register, the result register, is used to return information back to the computation after the operation has been performed. For example, if the computation was locking a semaphore then the result register might be updated to indicate that the lock could not be obtained. A more thorough description of the HWTI's state machine can be found in [3, 1].

3. INTERMEDIATE FORM

The second layer in the system is the intermediate form layer. This layer provides a bridge between high level languages and the hthreads operating system. This is accomplished by divorcing the semantics of a computation from its implementation strategy. The main benefit of this is that many high level languages can be supported along with many different implementation strategies without exponentially increasing complexity. Creating such an intermediate form, however, is a difficult task. Consequently, our initial version of the hthread's intermediate form is a more restricted version. This version only aims to support the C programming language along with an implementation strategy based on finite state machines. We refer to this restricted version of the intermediate form as the hardware intermediate form (HIF). The HIF is currently capable of transforming its intermediate form syntax into a state machine implementation by way of VHDL.

The HIF is a conglomeration of a RISC-like instruction set with high level abstractions for control flow. The RISC-like instruction set provides arithmetic and memory operations to the intermediate form. These instructions are standard 3-tuples. The major difference between the HIF and standard RISC instruction sets is that the HIF does not use a canonical register set. Instead the HIF makes use of generic variables. The only restriction on these variables is that they are always implemented as 32-bit values. The HIF does not make any distinction between shared memory and local memory because the HWTI is capable of resolving the difference during execution. The high level control flow operations supported by the HIF are goto statements, if state-

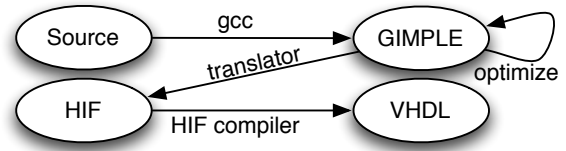


Fig. 1. Compilation Flow

ments, case statements, and function calls. The function call model of the HIF supports both efficient tail recursion and more expensive but more powerful generalized recursion. The simplicity of the HIF is an artifact of the underlying support provided by the HWTI. The HWTI abstracts much of the complexity of the hthread operating system and frees the HIF to focus on support for the computational model provided.

The current implementation of the HIF compiler is still under development. This version, however, is capable of translating most of the intermediate form into VHDL state machines. These state machines can then be synthesized using standard tools and executed inside of reconfigurable fabric.

4. HIGH LEVEL LANGUAGES

The last layer is support for high level languages. This layer builds upon the intermediate form layer and completes the bridge between typical programmers and reconfigurable devices. This is done by transforming high level languages into the intermediate form. This intermediate form is then transformed into VHDL which is then synthesized using standard tools. Ideally this layer would include many different high level languages. Supporting many high level languages is an arduous task and for this reason we have concentrated our initial development on only a single high level language, the C programming language.

The design of our high level language translator is based around C code which uses the hthreads libraries along with GCC's GIMPLE intermediate form [4]. It is important to note that the hthreads libraries are not special additions to the C programming language but are standard C libraries. Figure 1 shows the overall compilation flow of hthreads. Compilation starts with standard C source code. This is translated into the GIMPLE intermediate form by GCC's frontend. Once in this form, GCC will apply many semantic-preserving optimizations to the code. After this is complete, our translator transforms the GIMPLE intermediate form into the HIF. The HIF compiler is then capable of transforming the HIF into VHDL for synthesis. One interesting feature of the compilation flow is that GIMPLE is a common intermediate form used by all GCC-supported language. Thus, this compiler flow should be able to easily support any lan-

```

1  typedef struct { int x,y,a,r; } mac_t;
2  void* multi_acc( void *arg ) {
3      mac_t *mac = (mac_t*)arg;
4      mac->r = mac->x * mac->y + mac->a;
5      return arg; }

```

Fig. 2. Multiply Accumulate in C

guage which uses the GIMPLE intermediate form, though we are focusing our efforts only on supporting the GIMPLE features which the C frontend uses.

The initial implementation of the C to HIF translator is currently being developed. The current version, however, is capable of transforming many features from the C programming language into HIF. Once in its HIF form, these features can eventually become hardware by way of the HIF compiler.

5. EXAMPLE

To demonstrate the hthreads compilation flow consider the source code show in figure 2. This source code contains only a single function, `multi_acc`. The HIF translation of this function is shown in figure 3. This HIF shown is output from the current version of the HIF compiler. Note that the simplicity of this example is intentional and that the hthreads compiler is capable of handling code which is much more complex.

A function of one argument named `multi_acc` is declared on the first line of the HIF. This corresponds to the function definition on line 2 of the source code. Line 2 in the HIF reads in the function's argument and assigns it to the variable `hif_arg`. The third line then renames this argument to `hif_mac` and corresponds to line 3 in the source code. Lines 4 through 9 are a result of line 4 in the source code. These lines read in two values from memory and multiply them. They then read in a third value from memory and add this value to the multiplication. Line 9 in the HIF stores the results to memory. The last line in the HIF returns the result of the function and corresponds to line 5 in the source code.

There are several features of the HIF translator which are important. First, the HIF translator is capable of translating C into HIF without any need of special syntax or language augmentations. Second, the generated state machine runs in true parallel with any other computations which are executing. This provides coarse-grained parallelism in the system and frees the translator and compiler from needing to extract the maximum level of instruction level parallelism from the source code.

```

1  function multi_acc 1
2      readarg hif_arg 0
3      hif_mac <- hif_arg
4      read x hif_mac 0
5      read y hif_mac 4
6      tmp1 <- mul x y
7      read a hif_mac 8
8      tmp2 <- add tmp1 a
9      write hif_mac 12 tmp2
10     return hif_arg

```

Fig. 3. Multiply Accumulate in HIF

6. CONCLUSION

This paper has presented the goals, the design, and the implementations of the unified computational model known as hthreads. Hthreads has the lofty goals of raising the programming capabilities of massively parallel reconfigurable devices to be in alignment with the computational capabilities of those devices. The paper has presented the three layers of abstraction which are provided on top of standard reconfigurable devices.

The development of all three of these layers is ongoing but initial versions of each layer have already been completed. Once fully completed, this system will be capable of bringing high level programming support to the reconfigurable systems domain.

Acknowledgment

The work in this article is partially sponsored by National Science Foundation EHS contract CCR-0311599.

7. REFERENCES

- [1] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbee, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid fpga-cpu computatoinal components: A missing link," *IEEE Micro*, vol. 24(4), pp. 42–53, 2004.
- [2] D. Andrews, D. Niehaus, and P. J. Ashenden, "Programming models for hybrid cpu/fpga chips," *IEEE Computer*, vol. 37(1), pp. 118–120, 2004.
- [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," in *Proc. of FCCM'06*.
- [4] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *GCC Developers Summit*, 2003, pp. 171–180.
- [5] M. Vuletic, L. Pozzi, and P. Ienne, "Seamless hardware software integration in reconfigurable computing systems," *IEEE Design and Test of Computers*, pp. 102–113, 2005.