MultiCore eMIPS

Zhimin Chen, Richard Neil Pittman, Alessandro Forin
Microsoft Research

August 2009

Technical Report
MSR-TR-2009-113

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

MultiCore eMIPS

Zhimin Chen, Richard Neil Pittman, Alessandro Forin
Microsoft Research

ABSTRACT

In this work, we propose a combined multi-core architecture with the capability for instruction set extension (ISE). We show that while both multi-core and ISE exploit parallelism, they do so differently. For this reason multiple cores and ISE could be combined to obtain greater performance improvement than the sum of the two alone. To evaluate this, we implement a dual core microprocessor on a FPGA using an extensible soft-core, eMIPS and mapped the Floyd-Warshall all points shortest path algorithm to this system. Overall, the combined technique yielded performance 3.25x faster and 1.73x faster than parallel or ISE techniques alone.

1. Introduction

The advancement of microprocessor performance based on increasing clock frequencies into the higher gigahertz range has come to an end. Modern technology and manufacturing processes cannot overcome the stability and thermal issues at these higher frequencies. New fabrication processes however have made it possible to make denser logic at the same or reduced cost. The availability of additional computing resources has made parallel execution on multiple functional blocks or processing units a promising new vector for microprocessor advancement.

These parallel execution blocks have taken the form of additional general purpose processor cores, application coprocessors and instruction set extensions. These additional processing units can perform tasks in parallel increasing throughput and performance and in some cases more efficiently than the general purpose microprocessor cores they support. Additionally, these processing units can be specialized to perform functions not previously available to general purpose microprocessors.

Combined with their respective programming models and APIs, these new architecture exploit different types of parallelism that exists within user applications with varying levels of success. As these different architecture features for improving performance by exploiting parallelism emerge, we should recognize that these techniques are not mutually exclusive. These techniques could be combined without one detracting from another and in some cases the combination can be more than the sum of its parts.

This is the case we make for extensible instruction set multi-core microprocessors. In this work, we present the design of a extensible instruction set multi-core microprocessor based on previous work done with the extensible MIPS RISC microprocessor, eMIPS [7]. Then we will demonstrate how better performance can be achieved combining extensible processor cores into a single microprocessor than either multiple cores or instruction extension alone.

2. Multi-Core versus Instruction Set Extension (ISE)

The multi-core revolution occurred in the convergence of two important circumstances. First, hardware engineers have pushed the frequency and thermal capabilities of modern circuit technology. Second, improvements in fabrication processes have significantly reduced the sizes of transistors that make these circuits, making it possible to pack orders of magnitude more logic density into the same die.

Microprocessor developers can now manufacture more of their existing architectures smaller and more cheaply with these new processes. However, since they could not find a way to clock them any faster they could not continue the aggressive progress in processing power from generation to generation to which they had become accustomed. The solution that emerged was to pack more these smaller, cheaper cores into a single microprocessor chip.

Multi-core microprocessors are now becoming ubiquitous in all types of computing. Sony, Toshiba and IBM produced their well-known CELL processor which contains a single Power Processor Element and eight Synergistic Processor Elements [1]. NVIDIA's multi-core graphics processor units (GPUs), e.g. 128-core GeForce 8800, provide another example of multi-cores in graphics applications [2]. General purpose processor providers, Intel and AMD, have their own multi-core architectures represented by the Xeon [3] and Opteron [4] respectively.

The ideal maximum improvement in performance a multi-core microprocessor provides over a single core is directly proportional to the number of cores used to perform the same task. In other words, a microprocessor with X cores should be at most X times faster than a microprocessor with a single similar core. However, software developers

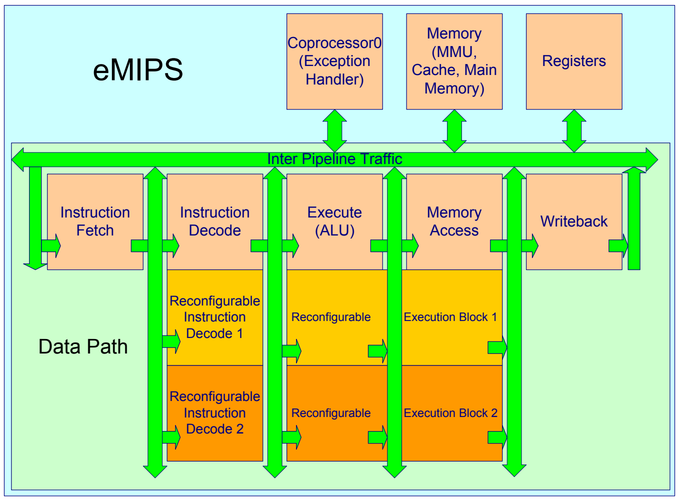


Figure 1. eMIPS architecture [7].

continue to have difficulty taking full advantage of the computing power modern multi-core microprocessors provide. This shortcoming is represented in Amdahl's Law, which states that there exists a limit to the performance improvement provided by multi-core microprocessor on applications where only a portion can be parallelized. In addition, the degree of parallelism within these portions of the application is a factor [5].

If the application can only be partitioned into two, the most improvement a multi-core implementation can expect on a multi-core system with more than two cores is 2x assuming this is true for the entire application. If this is true for only half the application this performance improvement further degrades to 1.34x. These approximations do not account for the effects of other factors including cache architectures and memory speeds. They do not account for overheads for partitioning and synchronization required in multi-core systems either.

Instruction set extension has gained popularity in the embedded space where microprocessors with reduced or minimal capabilities are augmented with specialized instructions for a particular application to conserve area, power and cost with relative minimal impact on performance. These specialized static microprocessors are the predecessor of a new trend, configurable computing where dynamically extensible microprocessor can configure specialized hardware to augment capabilities and application performance at runtime.

Our own eMIPS microprocessor is an example of a dynamically extensible processor [7]. Figure 1 provides a high level block diagram of the eMIPS extensible microprocessor. There are several commercial examples of statically extensible or custom microprocessors such as the Tensilica Xtensa [19] , the MIPS Pro Series [20] and the ARC 6000 Series [21].

Rather than partitioning out large task to be carried out in parallel like multi-core, instruction set extension augments the capabilities of a single core by making highly repeated and common task more efficient. In some cases this is done by taking advantage of low level parallelism in the operations or data of the application's execution. Like the multi-cores, the instruction set extension performance is bounded by Amdahl's Law [5]. The performance improvement provided by an instruction extension is derived by the proportion of execution time the instruction extension is in use times the speed up using the instruction extension over using the general purpose hardware. Therefore, if an instruction set extension can perform a task 2x faster than the general purpose hardware but the task only makes up half the execution time, the overall speed up is only 1.34x like the multi-core.

However, unlike the multi-core scenario, the use of dynamic instruction set extensions to accelerate application performance requires a level of hardware knowledge that most software developers simply do not possess. In the past tools for assisting software developers with this have been difficult to use and variable in their effectiveness. This has improved in recent years but they still have much further to go before the average software developer will consider them. Despite this, the demonstration of some computational and data intensive applications attaining orders of 1000x speed up using instruction set extension and dedicated hardware keeps interest alive.

Given an application where at least half can be partitioned between two processing cores. Lets propose that both partitions of this code contain tasks that can be optimized using instruction set extensions to increase performance of those task like the previous paragraphs. By combining these techniques on this part of the application, the overall performance could theoretically approach 1.6x. Then, if we could use instruction set extension on the part of the application that could not be partitioned, the application could come closer the 2x speed up you would expect from a dual core system.

3. MultiCore eMIPS Research Platform

We used the Berkeley Emulation Engine (BEE3) developed in cooperation with Microsoft Research and UC Berkeley for implementing the MultiCore eMIPS system. The BEE3 includes four Virtex 5 FPGAs (xc5vlx110t-2ff1136). Each FPGA has two channels of 8 GB of DDR2 memory for a total of 16 GB per FPGA and 64 GB for the entire system. Each FPGA also has a serial line and Ethernet. The FPGAs of the BEE3 are linked in a ring configuration to allow for high speed communication between them.

Development of the MultiCore eMIPS can be configured into two different ways: Single FPGA shared memory configuration and the Multi-FPGA message passing configuration. The following sections describe the design of these configurations in greater detail.

3.1 Single FPGA Shared Memory

The original eMIPS microprocessor from which this work is derived utilizes a memory mapped IO system for interfacing to memory and peripheral devices. This method is carried over into this phase of the multi-core development. In order to maximize flexibility within the system, we classified peripherals into two classes: local memory peripherals (LMP) and shared memory peripherals (SMP).

Figure 2 represents a organization of the single FPGA shared memory system with two cores. Given the size of the FPGA, resource requirements of the cores and supporting hardware interfaces we decided to include two cores to leave enough room to make implementation easier and add features later.

3.1.1 Local Peripherals

Local Peripherals are connected directly to the memory interface of their host core. Each core in the eMIPS MultiCore system has its own set of local peripherals and cannot access the local peripherals of any other core. For this reason the address spaces of these local peripherals can overlap. Local peripherals of each core can be assigned to the same address space and each core will only access their own. The set of local peripherals includes a local block ram and timer.

The local block ram is a memory that is fast and close to its host core. It provides the core with a memory space that it can use for performing local task without competing with other cores for the memory resource. The typical configuration is 32-bit wide, one cycle access time, and 64 kB bytes total. The local timer is used for local scheduling task. The host core can use either a free running counter or down counter to trigger interrupts or keep track of elapsed time. The timer uses a 10 MHZ clock which makes converting ticks from the timers to real time very easy.

3.1.2 Shared peripherals and the Bridge

The shared peripherals are those that can be used by either core in the single FPGA system. These peripherals either have a global effect that is used for synchronization or communication between the cores, or controls a singly instantiated, shared physical resource. These peripherals were designed for use with the original eMIPS microprocessors and thus interface to the local memory bus of one core. In order to allow two cores to share this peripheral, bridges were implemented for each peripheral to

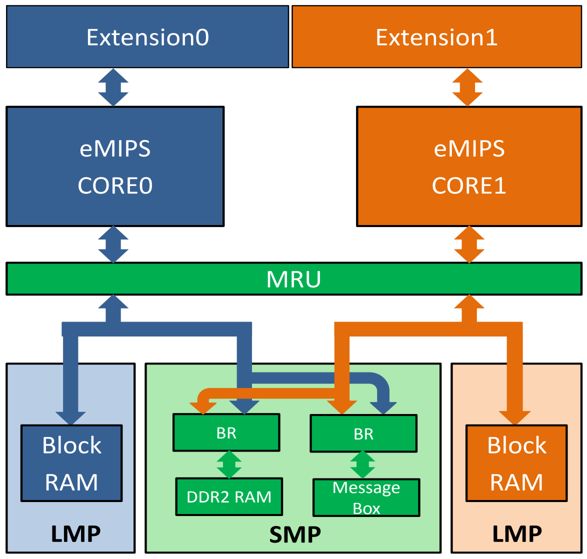


Figure 2. Structure of the Single FPGA shared Memory system.

connect them to the local memory bus of each core. The bridge implements the local bus protocol using three ports: one to each core and one to the peripheral. Memory commands enter through the core ports and are either routed to the peripheral port or held waiting depending on the free/busy state of the bridge.

In order to add and remove peripherals in an easy modular fashion, address decoding is performed distributed within the peripherals themselves. Each peripheral connects to the address bus of the memory interface. The peripherals evaluate the address on the address bus to determine if it is within range assigned to this peripheral. If so it acknowledges the request and performs the read or write transaction. Based on which peripheral acknowledges the request the memory interface routes data from the peripheral to the core or vice versa.

In order to design the bridge, an issue with the address decoding had to be solved. The problem is that the bridge does not include this address decoding logic and does not know the address range for the peripheral it is connected to. In order to maximize model reuse, we chose to implement the bridge as a series of multiplexor controlled by a state machine. The state machine implements a round-robin protocol for connecting the two core ports to the one peripheral port on each bridge.

The state machine contains six states including idle, listening and active for each core. Figure 3 provides a flow chart of the bridge state machine. States S00, S01 and S02 are mirror states to S10, S11 and S12 respectively. States S0x act on Core 0 and states S1x act on Core 1.

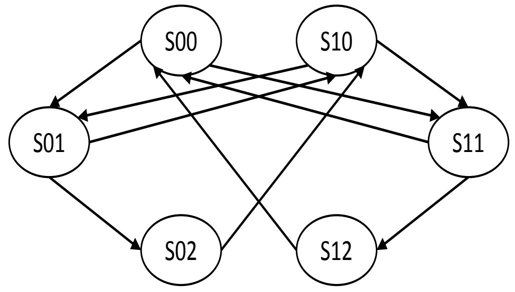


Figure 3. State Machine for the Distributed Shared Peripheral Bridges.

States Sx0 are idle states. During these states, the shared peripheral connected to the bridge is not connected to either core and the local bus connecting the peripheral to the bridge is inactive. While in states Sx0, the bridge waits for a memory request from the cores. If the state machine is in state S00, it gives preference to Core 0 and in state S10 to Core 1. When the bridge gets a request it transitions states to one of the states Sx1: state S01 for a request from Core 0 and state S11 for Core 1.

States Sx1 are the listening states. During these states, the bridge has received a memory request from one of the cores. In state S01 Core 0 is active and in state S11 Core 1 is active. The bridge routes the address of the memory request from the active core to the peripheral and waits for the peripheral to respond. If the address is within the peripheral's range it will send an acknowledgement. The bridge routes this acknowledgement to the active core and transitions to states Sx2: S02 for Core 0 and S12 for Core 1. If the peripheral does not respond within seven cycles or another peripheral responds, the bridge transitions to the Sx0 state of the core opposite of the active core: S10 for Core 0 and S00 for Core 1.

States Sx2 are the active states. During these states the bridge has received a memory request from a core and the peripheral has acknowledged it. In state S01 Core 0 is active and in state S11 Core 1 is active. In these states the data busses in and out of the peripheral are routed between the active core and the peripheral through the bridge. The bridge waits for the peripheral to signal the transaction is complete before disconnecting the core from peripheral and returning to the idle state opposite to the active core: S10 for Core 0 and S00 for Core 1.

Peripherals included in the shared peripherals include the DDR2 memory, serial line, Ethernet, global timer, shared block ram, interrupt controller, message boxes and processor id registers. Some of these peripherals provide essential functions for the multi-core system and will be discussed in more detail. The DDR2 memory, serial line and Ethernet are physical resources and must be shared by the cores.

3.1.3 Contention Resolution for Shared Memory and the Memory Reservation Unit (MRU)

In a single FPGA system, the cores share access to the memory resources including the shared block ram and the DDR2. In this situation it is possible for one core to modify the memory being accessed by another. Software can be written to avoid this situation but a hardware mechanism is still required to prevent collision. The MIPS ISA provides support for atomic read modify write that can be used for this purpose. This mechanism is realized via the instructions LoadLink (LL) and StoreConditional (SC).

LoadLink is a 32-bit word load that reads data from a given address into a general purpose register inside the MIPS core. In addition to reading the data, Loadlink reserves that address for that core. The reservation remains valid until another LoadLink is issued for another address or a store occurs on that address that is not a store conditional for that core.

StoreConditional is a 32-bit word store that writes the contents of a general purpose register to a given address. What makes this different from a normal store is that the store only happens if the address being written to has a valid reservation from a previous LoadLink. If the reservation is still valid, the store completes and a one is written into the general purpose register the data had come from. Otherwise the data is not written and the general purpose register is given the value of zero.

The value of the register after the StoreConditional communicates to software whether the store occurred or whether a collision has occurred. In this way, it is possible for a MIPS core to support lock free atomic read modify write across threads, processes, and cores on shared memory. This mechanism is implemented in the memory reservation unit (MRU).

The memory reservation works using physical memory addresses instead of virtual memory addresses. For this reason, in the original eMIPS implementation the MRU was placed in the memory data path after the memory management unit (MMU) but before the local memory bus to evaluate the reservation status as early as possible. However, to support two cores, it was necessary to move this element after the local memory bus and before the peripherals, as seen in Figure 2.

```
/* barrier function for core 0 */
  void barrier(void){
          volatile UINT32 * mb0 = MBADDR0;
          volatile UINT32 * mb1 = MBADDR1;
          *mb0 = 0x5555aaaa;
          while(*mb1 != 0x5555aaaa);
          *mb1 = 0x0;
  }

/* barrier function for core 1 */
  void barrier(void){
          volatile UINT32 * mb0 = MBADDR0;
          volatile UINT32 * mb1 = MBADDR1;
          *mb1 = 0x5555aaaa;
          while(*mb0 != 0x5555aaaa);
          *mb0 = 0x0;
  }
```

Figure 4. Barrier synchronization functions.

The MRU receives memory transaction request from both cores in parallel and evaluates them against any address stored in the reservation with a valid flag. MRU passes loads through to the local memory bus regardless. LoadLink instructions are passed through but the MRU updates the reservation and sets the valid flag. The MRU passes stores through but if the address matches the address in the reservation, the MRU clears the valid flag. If the MRU gets a StoreConditional instruction it evaluates the reservation address and the valid flag. If the address matches and the valid flag is set, the store goes through and the MRU notifies the core of the success. If the address does not match or the valid flag is clear the store does not go through and the MRU similarly notifies the core of the failure.

The MRU includes only one reservation for the two cores in the single FPGA system. Therefore, it is possible for the two cores to get into livelock condition if they repeatedly attempt an atomic read modify write at the same time that interleaves the LoadLink and StoreConditional instructions. Software mechanisms such as random delays on a failed StoreConditional are possible solutions to this situation.

3.1.4 Other Synchronization Mechanisms

The single FPGA multi-core system supports several mechanisms for core synchronization. It is necessary for the cores working on related problems to communicate in order to deal with data and control dependencies. In order to facilitate this communication, the single FPGA system includes the shared message boxes.

Each core in the single FPGA system has its own message box within the shared address space. In order to coordinate and synchronize, cores can send messages to each other using these messages boxes. The current implementation of the messages box includes a 32-bit register accessible through memory mapped IO where cores can write user specified command codes within their applications. Additional space could be added for more complex commands and communications.

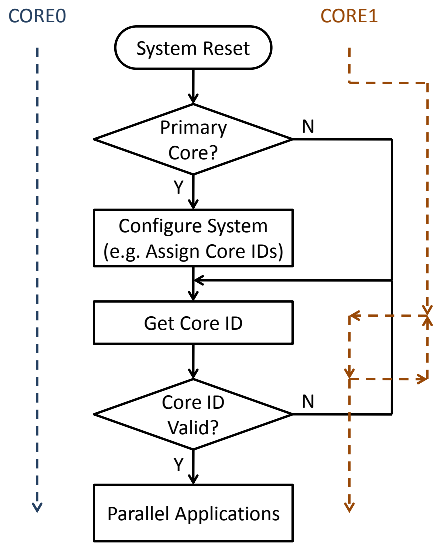


Figure 5. Dual core boot up and initialization process.

With these message boxes, we implement a simple barrier synchronization function for the two cores. At certain points in the application execution, the barrier function is called to force the core to wait on the other core. Inside the barrier function the core spins polling its message box. It is waiting for the other core to write the go message into its message box. Example code for these functions is provided in Figure 4.

In some cases, polling the shared message box would not be preferred because of the additional traffic on the shared peripheral. Instead the barrier could be modified spin on a global Boolean variable in the cores local memory. The state of this variable can be changed in an interrupt service routine triggered by an interrupt from the other core.

The shared interrupt controller allows the software developer to route interrupts from peripheral devices to one or both cores. The interrupt controller also allows each core to send interrupts to the other. In this way the barrier functions could be modified to have the active core write the go command to the idle cores message box and then trigger an interrupt to that core through the interrupt controller. The idle core stop spinning in the barrier function to service the interrupt. Since it is a neighbor core interrupt, the interrupt service routine will read the message box. If it sees the go command, it changes the state of the barrier Boolean variable and the idle core will exit the barrier function.

3.1.5 Initialization
When the microprocessor system starts up for the first time or reboots, it is necessary for the system to initialize itself and prepare to run application code. In the first eMIPS version this was done by running a short program stored in the on-chip block ram at the reset vector. With multiple cores however, this becomes a little more complicated. A flow chart of the initialization sequence is given in Figure 5.

To configure the peripherals, we read and write the configuration registers available to us through the memory mapped IO. If we naively allowed both cores to share the same start up code and had both start configuring the peripherals same time. This would result in these reads and writes occurring twice and in different orders. Some of these reads and writes have side effects that would result in invalid configurations if documented initialization is not performed correctly.

Since we cannot have both cores executing initialization code at the same time, we designate Core 0 to be the master core in the system and the only active core at system reset. This is accomplished using the processor id registers. Only the master core, Core 0, has a valid processor id and can start running. All other cores, such as Core 1, do not have valid processor ids and are considered inactive.

When the system comes out of reset, Core 0 has a valid processor id and Core 1 does not. Both Cores start fetching instructions from their reset vectors which point to their local block rams. Contained within the block ram is the processor initialization code. Both cores run this local initialization until they reach a processor id check. The check forces the processor to spin checking its processor id register until reads a valid processor id. Since only Core 0 has one, it is the only core that gets past this check and does not spin.

While Core 1 spins, Core 0 proceeds to initialize the system including initializing the memory system and the serial line. Then the Core 0 either activates Core 1 by writing a valid processor id to its register or it can put the core to sleep. At this time, the system enters the application code. Only Core 0 can write to the processor id registers of the cores on its system.

3.2 Multi-FPGA Message Passing
The single FPGA system currently contains two processing cores in a largely self contained system. This same design could be configured to all four of the FPGAs of the BEE3 and they could communicate over the Ethernet in cluster like configuration.

However, since the FPGAs on the BEE3 are linked using the ring, it is possible to link them into a more tightly coupled system of eight processing cores. To achieve this, the original dual core design is expanded to include a message passing router on the BEE3 ring as presented in Figure 6. These four pairs of cores will be more decoupled with their own memory than the pairs on the same FPGA that share memory. There are three considerations when setting up this system:

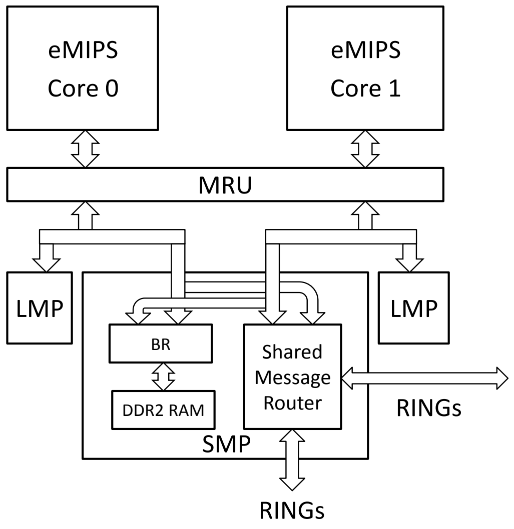


Figure 6. Structure of Single FPGA with Multi-FPGA Message Passing.

1) How to guarantee the low level signal stability transferring data on the ring?
2) How to route data from one processor core to another?
3) How to initialize the system across multiple FPGAs and cores?

The following sections will discuss these issues and present the solutions we developed to deal with them.

3.2.1 BEE3 Ring
The BEE3 ring connects the four FPGAs using point to point wired connections between a pair of neighbor FPGAs. Figure 7 shows how the ring connects the FPGAs on the BEE3 system. Each connection between a pair of FPGAs includes 72 bidirectional data lines and five bidirectional strobes. On top of this we constructed two 36-bit rings, 32-bit data and 4-bit parity, with some control signals each going in opposite directions. In this way, each ring is unidirectional which simplifies the design and implementation but maintains two way communication between each pair of FPGAs

There are two concerns that must be addressed in order to reliably use the ring for inter-chip communication. First, the trace delays of the wires that make up the ring connections are not guaranteed to be of the same length,

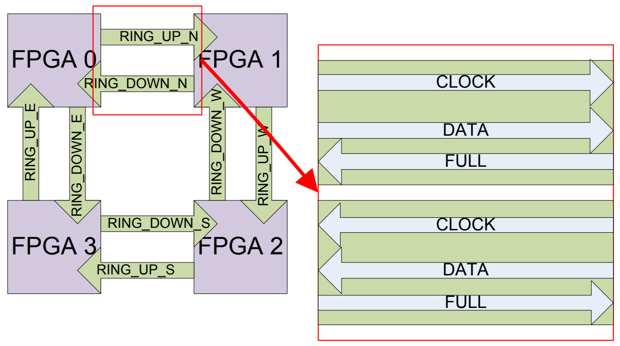


Figure 7. BEE3 Ring.

which skews signals across different wires. Even if the traces are very close, minor variations in e.g. temperature can have significant impact in how fast a signal propagates through one wire versus another. The second issue, is that we cannot know or guarantee that the clocks of any pair of FPGAs on the BEE3 are in any way aligned.

It is possible to align the trace delays of the data lines between the FPGAs, using the configurable input delays of the FPGA's configurable I/O pins. This would require a complex calibration procedure each time the system is reset, and recalibration could be required during system operation due to changes in temperature. The costs in engineering effort and FPGA logic resources to make it work correctly were considered too high. Given the short time frame for this work, and high speed communication between FPGAs is not the focus of this work, this approach was not taken.

If the frequency at which the data lines switch on the ring is sufficiently low the small difference in propagation delay become negligible compared to the overall time the data is valid. We developed and tested the communication frame work using a relatively low 50 MHz clock, and we achieved complete reliable communication between the FPGAs, under saturated conditions. We tested the ring using a 100 MHz clock as well, but the testing rig we designed was not able to saturate the ring at this frequency. At a frequency of 50 MHz, we were able to maintain a bandwidth roughly 1.6 Gb/s, versus 1 Gb/s for Ethernet due to the wider data width over a direct point to point connection. For this reason, we proceeded with the 50 MHz implementation.

We still have to deal with the lack of clock alignment between the FPGAs. The send and receive pins on each FPGA are implemented using FIFOs with independent read and write clocks. In addition to this, we used two of the five strobe lines in the ring to transmit the clock of the transmitter to the receiver. The transmitter fills the transmitter FIFO using its internal logic clock, at 100 MHz. The transmitter sends data from its transmit FIFO using its 50 MHz ring clock. This clock is sent over one of the

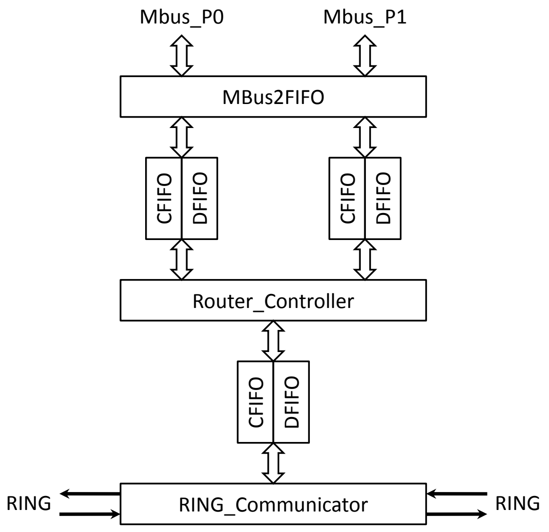


Figure 8. Message Packet Router.

strobe lines to the receiver and is used as the write clock input of the receiver FIFO. The receiving FPGA then reads the contents of the FIFO using its own clock, at 100 MHz. In this way the clock used to receive the data is aligned to the data, but does not need to be aligned with the receiver's main clock.

3.2.2 Message Packets and Routers

The messages the FPGAs send to each other on the ring are organized into packets. Each packet has four parts: lead in, header, payload, and lead out.

The lead in and lead out are patterns of bits appended to the beginning and end of the packet by the transmitter and stripped off by the receiver. The purpose of these is to communicate between the transmitter and the receiver what is the beginning and end of the packet currently in transmission. The transmitter first sends the lead in before sending the packet from the FPGA cores and sends the lead out after the last word of the packet. The receiver listens for the lead in on its ring inputs to activate its capturing and forwarding logic and listens for the lead out to reset it.

Immediately following the lead in is the packet header. The packet header is a single 32-bit word that contains four fields: 8-bit destination, 8-bit source, 6-bit control and 10-bit size. The destination and source fields contain the processor ids of the destination and source cores for the packet. The destination field is used by the routers to deliver the packet to the correct FPGA and core. The source is simply for record keeping so the core receiving the packet can know where it came from. The control field contains information about the packet including the direction on the ring it is travelling, whether it is a data or command packet and the count of hops the packet has already made. The direction control tells the transmitter which ring output to send the packet on. When the hop count reaches four without reaching its destination the receiver drops the packet as dead or undeliverable. This case will only happen if the value in the destination field does not correspond to an assigned processor id. Most packets sent on the ring are data packets. A small number of command packets are used to configure the communication on the ring at initialization. With the size field, the ring's transmitters and receivers can support packets up to a kilobyte in size.

The message packet router implemented on each FPGA has multiple levels as depicted in Figure 8. There is local routing between cores on the same FPGA without involving the ring. If the destination is outside the local FPGA it is dispatched appropriately on the ring.

Using a DMA interface, the source core writes the header to the command FIFO and the data to a data FIFO. The high level router control pulls the head off the command FIFO and evaluates the destination of the message packet. If the destination processor id is within the same FPGA as the source, the header and data are forwarded to the receiving command and data FIFOs of the destination core. Similarly a DMA operation transfers the header and data to memory before interrupting the receiving core.

If the packet is bound for another FPGA, it arrives at the transmitter where the direction control bit determines the ring transmit FIFO the packet is written to. The transmitter first sends the lead in, then the header, data and finally the lead out. When the packet is received by the neighbor FIFO the destination processor id is evaluated against the processors ids of the cores on that FPGA. If it is a match for any of them, the packet is stripped of the lead in and lead out as it is forwarded to the appropriate core by the high level router control. However, if this is not the destination FPGA for the packet, the hop count is incremented before forwarding the packet to the next FPGA in the ring through this FPGA's transmitter. If the hop count manages to get to four without being delivered, it is dropped.

3.2.3 Multi-FPGA Initialization

As was discussed in section 3.1.5, the task of system initialization at start up and reset can be complicated by the addition of additional processing cores that can access peripheral hardware devices and that share software. This scenario is further complicated by having multiple cores residing on different physical FPGAs with their own clocks, configuration registers, memories and hardware peripherals. To handle this we extend the model we previously used for the multiple cores on a single FPGA.

Similarly to the single FPGA case we use the processor ids to control the flow of the initialization sequence. In the Multi-FPGA system there is one master core for the whole

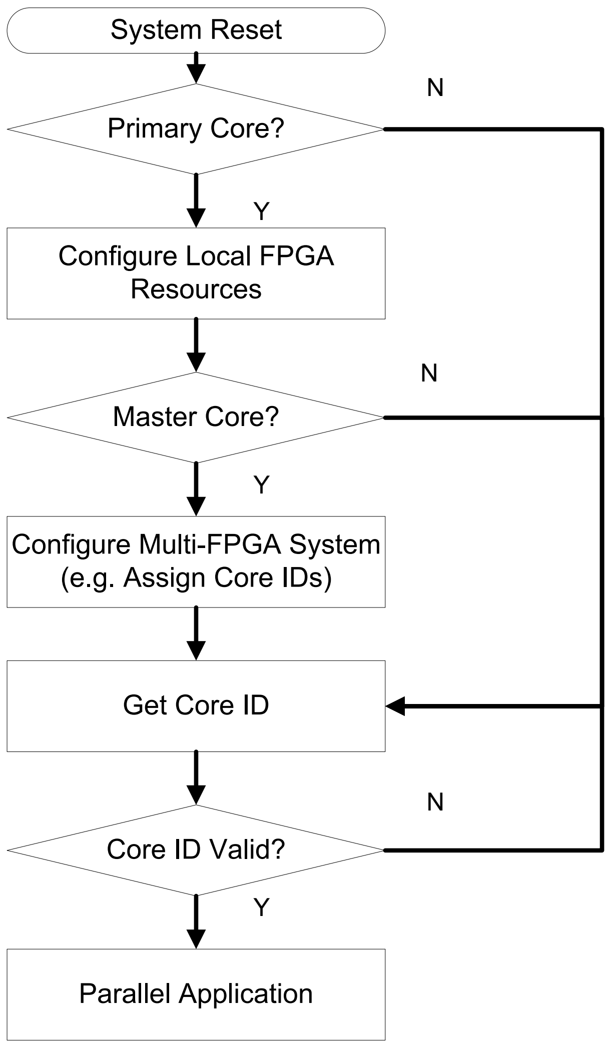


Figure 9. Multi-Core boot up and initialization across multiple FPGAs.

system that will coordinate the others, like the single FPGA system. In addition, each FPGA will have a primary core of its own. In the case of the FPGA with the master core, the master core is the primary core. Figure 9 provides a flow chart of the boot up and initialization sequence for the larger multiple FPGA system.

Here the most significant bit of the processor id is used to designate the processor id as valid or invalid. If the most significant bit of the processor id is set, it is considered invalid. The least significant bit of the processor id distinguishes the primary core of a single FPGA from the other cores in the same FPGA. This bit of the processor id is read only to the master core like the rest of the processor id is read only to the other cores. If the least significant bit of the processor id is clear, this is the primary core. Otherwise it is one of the secondary cores on the FPGA. The master core for the system is initialized with the processor id of zero (0x0), making it both a valid and primary core. The other primary cores are initialized with the processor id 0x80 and other cores are initialized to 0x81.

First, each core checks its processor id to determine if it is a primary core. If so, it initializes the local resources of that FPGA including the memory. Other resources can be initialized here as well or left for the application software to configure. At this point all cores check if they are the master core. Since Core 0 on FPGA 0 is the only master core, it proceeds to configure the serial line for console communication and any other initialization required for the primary core and FPGA that was not already done. The serial lines and Ethernet resources of the other non-master FPGA can be made usable by the application software if necessary. Meanwhile, the other cores spin polling their processor ids to see when they become valid.

To activate a core, the master core must change processor id that the core is initialized to a unique valid one. In the single FPGA shared memory implementation this was easily done through memory mapped IO with master core writing the processor id to the register. However, with multiple cores on different FPGAs that do not all share the same memory subsystem as the master core, this is not sufficient. Using the message passing capabilities of the ring the master core begins activating the other cores in the system.

As stated in section 3.2.2, there are two types of message packets used in the eMIPS MultiCore system: data and command packets. The master core makes use of these command packets to assign processor ids and activate other cores in the system. Control logic is included in the ring message packet router to update the processor id of a core when a command packet for assigning a processor id arrives.

Using the message passing infrastructure for configuring the system seems like the obvious solution. However, the message routing utilizes the processor ids to route the message to the intended core. So, how do we send a packet to a core to assign its processor id if it does not already have one? The solution is in the processor id assignment protocol implemented with the command packets.

The command packet for assigning a processor id is a single word header with no data payload. The source processor id field contains the processor id of the master core, 0x0. This is required for the command packet because only the master core can assign processor ids. The destination field of the header contains the new processor id to be assigned to the core. The command bit in the control flags is set and the direction bit is arbitrary. The size field is set to 0x0.

The protocol establishes that any core that does not have a valid processor id will receive this packet and the router control logic will assign the core the processor id in the destination field. After the processor id is assigned, the master core and any other core may send data packets to the core for synchronization or communication purposes.

There are no hardware mechanisms to prevent duplicate processor ids. It is the responsibility of the application software to assign and account for the processor ids in use. Consequently, software could choose to realize other, more complex schemes that do not assume the master is chosen by hardware.

3.3 Multi-Core Software

The current implementation of the eMIPS MultiCore system does not have any operating system support. The base ISA of the eMIPS MultiCore system is the MIPS R3000 ISA and to our knowledge there is no publicly available operating system for supporting multiple cores on a MIPS.

For this reason, we are limited to testing the system using custom test applications. We have written shared memory test programs to test the hardware features of the single FPGA and Multi-FPGA systems. This includes tests of the shared memory peripheral bridges, message boxes, processor ids, interrupts, synchronization, initialization and message routing.

Based on these tests using the barrier functions for synchronization we developed two test applications for the single FPGA system including a parallel Montgomery Multiplication and parallel Floyd-Warshall Algorithm. A case study using the Floyd-Warshall Algorithm to analyze the benefits and trade-offs of multi-core, ISE and the combination of the two is present in section 4.

At this time, the hardware support for the Multi-FPGA message passing has been completed, however software for using the message passing in an application is still required. As a result, all performance data collected and presented in this work reflect the single FPGA system using shared memory and barrier synchronization.

4. A Case Study on the Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm (FW) is a popular algorithm that solves the all pairs shortest path for a directed weighted graph with no negative cycles. The Floyd-Warshall Algorithm is an example of dynamic programming using an NxN matrix to represent the graph where N is the number of vertices in the graph. In order to evaluate our extensible multi-core system we implement four versions of this algorithm and gather performance data on each. We implemented a sequential version as our base line, a dual core version to run on our single FPGA multi-core system, an instruction set extension version using a hardware

```
1.  for i = 1 to N
2.    for j = 1 to N
3.      if there is an edge from i to j
4.        dist[i][j] = length(i,j)
5.        pred[i][j] = j
6.      else
7.        dist[i][j] = INFINITY
8.        pred[i][j] = NULL
9.
10. for k = 1 to N
11.   for i = 1 to N
12.     for j = 1 to N
13.       dist2 = dist[i][k] + dist[k][j]
14.       If(dist[i][j] > dist2)
15.         dist[i][j] = dist2
16.         pred[i][j] = k
```

Figure 10. Floyd-Warshall Algorithm.

accelerator and dual core version also using a pair of hardware accelerators (one in each core).

A pseudo-code description of the FW is presented in Figure 10. Both the dual core and the ISE versions of our implementation exploit the parallelism within the inner loop of the three deep for loop in lines 12 through 16. In this way, the FW differs from our hypothetical application described in section 2. The part of the FW that is targeted by both the dual core and ISE occupies over 95% of the application execution for sufficiently large graphs and approaching near 100% as the size of the graph grows. The remaining 5% or less is required for the initialization of the NxN matrix representation of the graph. Since we have very little going on except the computation we are interested in accelerating, the effects of the optimizations are more pronounced. This makes the FW a good candidate for evaluation of our strategy to use both techniques.

4.1 Sequential Version (Baseline)

Since both the dual core and ISE techniques attempt to exploit parallelism within a computational task to improve performance it is important to have a fair point from which to compare the two. For this reason, we implement a sequential version of the FW to run on a single base core with no hardware optimizations to assess the amount of work required to solve this shortest path problem for a given graph size, in our case thirty vertices.

FW finds the shortest path between all N vertices in a weighted, directed graph. In the first part of the algorithm given in Figure 10 (from Line 1 to Line 8), the *dist* and *pred* matrixes are initialized with the edge weights of the direct connections between all vertices and the destination vertex number, respectively. If no connection is found between two vertices, the values infinity and null are used. The second part of the algorithm is a 3-level nested loop. It

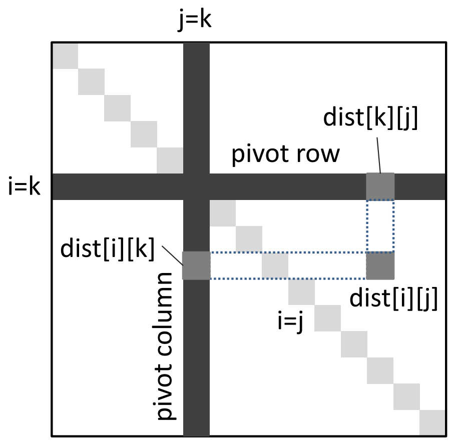


Figure 11. Sequential Floyd-Warshall Algorithm.

iteratively updates *dist* and *pred* with shortest path weights and the next vertex in the shortest path using N^3 comparisons. In detail, the basic operation within this part of the algorithm is to update the *dist* and *pred* matrixes for N times. Each time, the algorithm chooses a pivot row and a pivot column in *dist* (as shown in Figure 11). For each element of the matrix *dist[i][j]*, the program compares it with the sum of *dist[i][k]* and *dist[k][j]* which reside in the pivot column and pivot row. If the current *dist[i][j]* is larger than this sum, *dist*[i][j] is updated. Accordingly, k is stored to *pred*[i][j]. Looking at this algorithm, we can see that the pivot row and pivot column remains the same during each set of N*N matrix updates.

4.2 Dual Core Version

To investigate the performance benefits of partitioning an application across two processing cores, we apply a parallel programming scheme to the FW. The dominant computation within this application is the inner loop of the

```
// If core ID matches CORE0
1.  for k = 1 to N
2.    for i = 1 to N
3.      for j = 1 to i-1
4.        dist2 = dist[i][k] + dist[k][j]
5.        if(dist[i][j] > dist2)
6.          dist[i][j] = dist2
7.          pred[i][j] = k
8.    barrier();

// If core ID matches CORE1
9.  for k = 1 to N
10.   for i = 1 to N
11.     for j = i+1 to N
12.       dist2 = dist[i][k] + dist[k][j]
13.       if(dist[i][j] > dist2)
14.         dist[i][j] = dist2
15.         pred[i][j] = k
16.   barrier();
```

Figure 12. Parallelized Floyd-Warshall Algorithm.

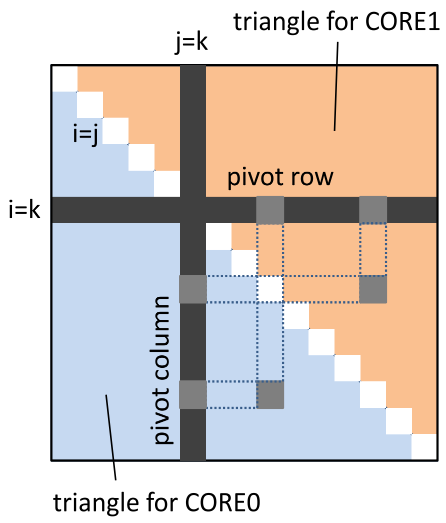


Figure 13. Parallel Floyd-Warshall Algorithm.

second half of the algorithm presented in Figure 10 (from Line 10 to Line 16). For this reason we focus our efforts at parallelizing the FW in this part of the algorithm.

In order to design an effective parallel programming scheme for the FW, we must identify a way to partition the work load of the algorithm in two parts as equally as possible. In addition, we need to minimize data dependencies between the two parts. Then we need to assign these parts to the two available cores.

After careful consideration of the FW we designed the following parallel programming scheme. Since the matrix elements where i=j are always 0, there is no need to handle them. As shown in the previous sections, in order to evaluate each element *dist*[i][j] and *pred*[i][j] requires current element, *dist*[i][j], and a element on the pivot column and pivot row, *dist*[i][k] and *dist*[k][j] respectively. The k^{th} pivot column and pivot row are not updated during k^{th} iteration of the outer loop. Therefore each element in the NxN matrix can be evaluated independently, for each k^{th} iteration. Given these observations we partition the FW along the i=j diagonal.

The pseudo-code description of the second half of the FW using this parallel programming scheme is given in Figure 12. When comparing this to the second half of the pseudo-code previously in Figure 10, attention should be given to the boundary conditions of the inner loop: Line 12 in Figure 10 and Lines 3 and 11 in Figure 12. These boundary conditions exclude the i=j diagonal itself and partitions the matrix along this line between the two cores. This produces two triangular regions of the NxN matrix that can be processed in parallel as presented in Figure 13: one with i<j and the other with i>j.

For each iteration through the matrix, CORE0 updates first triangle (i<j) and CORE1 handles the other (i>j). When each core finished processing its own triangle , it enters the barrier function presented in Section 3.1.4. The barrier function synchronizes the two cores at the end of each outer loop where the pivot column and row selections are updated. Both cores must remain synchronized to the same iteration of the outer loop because the $(k+1)^{th}$ iteration of each loop requires the k^{th} iteration in both cores to be complete and updated matrix data available.

To map the parallelized FW to our dual-core platform, we store the parallelized FW program instructions in the local BlockRAM memories of each core. This eliminates all memory conflicts due to instructions fetching, and optimizes instruction fetch time, effectively caching the whole program in high speed memory. The matrixes *dist* and *pred* are stored in the shared DDR2 SDRAM.

This parallel programming implementation has three major positive features:

1) The numbers of elements that need processing are the same for the two eMIPS cores. In total, each eMIPS core needs to process N*N*(N-1)/2 elements.

2) The two eMIPS cores are loosely coupled. To update one element, we only need the element itself and another two elements in the pair of pivot row k and pivot column k. The eMIPS cores only need to be synchronized at the end of each iteration of these loops indexed by k. In total, the number of synchronizations is N.

3) As for the control synchronization cost, we consider it very small because between two successive barrier synchronizations, each core should process one triangle (N*(N-1)/2 elements). Assuming every element has a similar probability to be updated, the task loads for the two cores between two synchronizations should be similar.

Based on the above analysis of the high level software implementation, it appears that the proposed parallel programming scheme is very symmetric and loosely coupled between the two cores. Therefore, a nearly linear speedup (close to 2) should be expected from the dual-core system using this algorithm.

4.3 ISE Hardware Accelerated Version

The eMIPS microprocessor core is a dynamically extensible microprocessor. As such, it has the capability to dynamically reconfigure a portion of its logic to support ISE. These ISE can be used to improve the performance of applications as well. We implement an ISE for the FW using the design flow described in [7] [24] [33].

By inspecting the pseudo-code representation of the algorithm in Figure 10, we can reduce the whole algorithm to a repetition of the following steps: load, add, compare,

```
05c4: lw a0,0(a3)        Load dist[i][k], dist[k][j],
05c8: lw v1,0(t2)        and dist[i][j] from DDR2.
05cc: lw v0,0(a1)
05d0: addiu t0,t0,1
05d4: addu a0,a0,v1      Compare dist[i][j] and
05d8: sltu v0,a0,v0      dist[i][k] + dist[k][j].
05dc: beqz v0,05ec
05e0: addiu a3,a3,4
05e4: sw a0,0(a1)        Update dist and pred
05e8: sw t3,0(t1)        matrixes with new values.
05ec: li v0,30
05f0: addiu a1,a1,4      Increase loop index, jump
05f4: bne t0,v0,5c4      back if boundary has not
05f8: addiu t1,t1,4      been met.
```

Figure 14. Hot Block for Sequential Floyd-Warhall Algorithm in MIPS Assembly.

conditional branch, store, compare, and conditional branch. These steps are encapsulated in the inner loop of the second half of the algorithm (from Line 12 to Line 16). This observation is borne out by the profiling results that identified this loop to be the most executed basic block of the FW. This basic block is referred to as the hot block. The MIPS assembly representation of this hot block identified by profiling the application in simulation is given in Figure 14. The hot block contains fourteen instructions that will be combined into a single instruction requiring multiple clock cycles, but in total less than the fourteen cycles required by the original code block.

An ISE in eMIPS is implemented as an additional processing data path tightly coupled with the MIPS data path. We use a Finite State Machine (FSM) to control the execution of the ISE. In the beginning, we used one state for each instruction in Figure 14. After that, optimization was performed to process several instructions without data dependencies in one state. Figure 15 illustrates all the states that the FSM goes through. Every level represents one state. Circles in each state represent the assembly instructions handled in that state. States with white circles cost one clock cycle while those with shaded circles cost multiple clock cycles. In detail, this processing procedure consists of 5 steps.

1) *Activation.* Decode instruction to determine if it is an Extension instruction. If instruction activates the Extension, it petitions the data path to execution.
2) *Request inputs.* Copy input registers from general purpose register file to local registers.
3) *Processing instructions.* FSM performs the semantic functions of the instructions in the basic block in parallel where possible.
4) *Write Back.* All registers that have changed state during execution are written back to the general purpose register file.

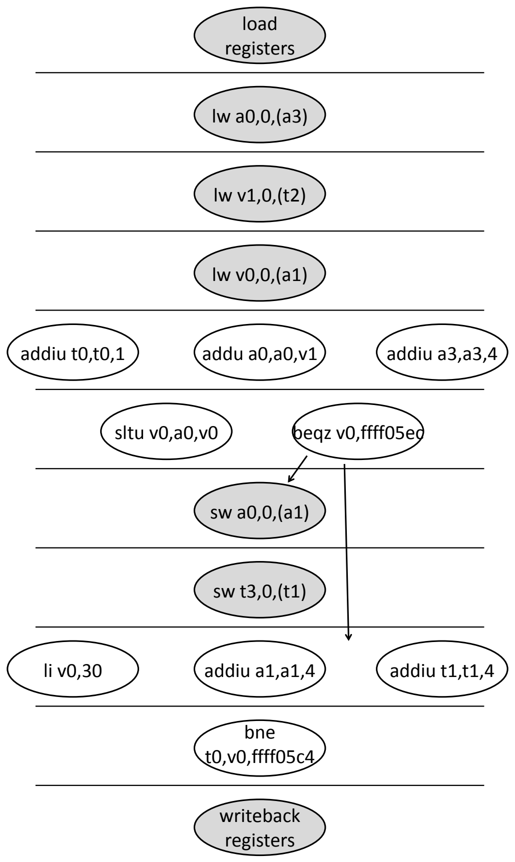


Figure 15. ISE for Sequential Floyd-Warshall.

5) *Resume Pipeline.* The Extension writes to the pipeline stage it is returning to, in this case WB. The Extension removes the stall and the pipeline resumes normal execution. The Extension instruction is committed in the WB stage.

The Extension watches the instruction decode path of the data path. It evaluates each instruction as it returns from memory to determine if it is an Extension instruction. If it is, the Extension petitions the data path for permission to execute. If permission is granted, the Extension signals the pipeline to stall while it is executes and updates the PC in IF to the address of the instruction immediately following the hot block.

The Extension fetches six registers from the general purpose register file: *a1*, *a3*, *t0*, *t1*, *t2* and *t3*, The registers *a3*, *t2* and *a1* are pointers to the values *dist*[i][k], *dist*[k][j] and *dist*[i][j] in the NxN matrix respectively. The register *t1* is a pointer to *pred*[i][j]. The registers *t0* and *t3* are the loop counters *j* and *k* respectively.

The Extension is implemented using a FSM that executes the functional semantic of the instructions in hot block in parallel as much as data and control dependencies will allow. First, the FSM performs three loads to fetch data from the NxN matrix in memory. The eMIPS Extension only has one memory port and these require several cycles to perform. For this reason these loads are executed sequentially. These loads can also result in a TLB Miss. Mechanisms for mitigating interrupts and exceptions in Extension execution are described in [7]. Next, the FSM performs three additions. Since these additions have no data or control dependency between them they can be executed in parallel. These instructions add *dist*[i][k] (*a0*) and *dist*[k][j] (*v1*), increment *j* (*t0*) and the *dist*[i][k] pointer (*a3*) respectively. Then the FSM compares the sum (*a0*) of *dist*[i][k] and *dist*[k][j] to *dist*[i][j] (*v0*) and branches if it is greater. If the branch is not taken, the sum (*a0*) is stored to the location of *dist*[i][j] (*a1*) and *pred*[i][j] (*t1*) is updated with the value of *k* (*t3*). Then, a constant for the boundary condition is loaded, thirty, and the pointers for *dist*[i][j] and *pred*[i][j] are incremented. The counter *j* (*t0*) is compared with the boundary condition and if it is less, the FSM updates the PC to the address of the Extension Instruction to loop back on itself. Otherwise, the PC remains the address of the instruction that immediately follows the hot block as it was set during activation. Finally, the FSM writes all registers that have changed state back to the general purpose register file including *a0*, *a1*, *a3*, *t0*, *t1*, *v0*, and *v1*.

The Extension pipeline synchronization logic reads the state of the FSM when it is complete and writes the PC of the Extension instruction and all pipeline state information to the WB pipeline stage of the data path. The Extension releases the stall on the data path, and the data path resumes execution normally as the Extension instruction commits in WB. If the Extension instruction loops back on itself, the PC is that of the Extension instruction. The Extension instruction will be fetched again and the process will repeat until the boundary conditions are met.

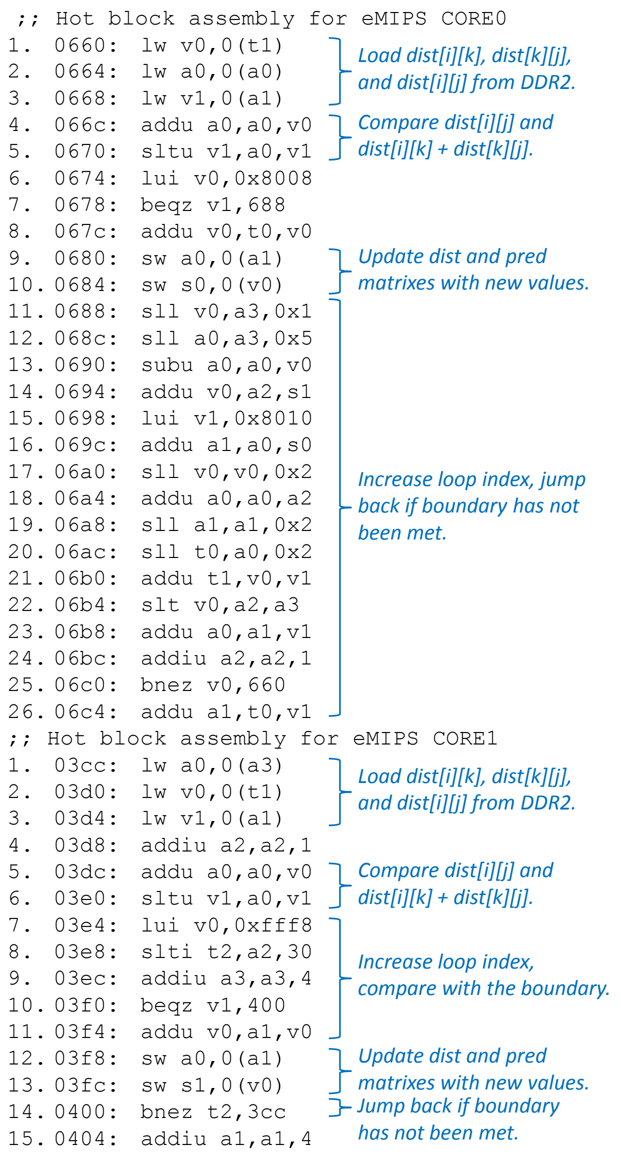
Compared with pure software execution, our ISE solution wins on three aspects. First, there is no instruction fetch operation. Second, different instructions without data dependencies are processed in parallel. Finally, intermediate register values are not written back to the register file during the intermediate stages. Only the final value of the register at the end of the execution is written back.

Our way to exploit parallelism with ISE is very straight forward. Several other methods or techniques can be used to obtain better solutions. For example, we can integrate a self-loop technique that would allow all the operations to stay in the ISE until the loop ends [33]. This would save operations to load and write back general purpose registers after each iteration. Looping within the hardware extension also saves synchronization overhead required to activate the extension hardware, stall the processor, release the pipeline and activate the extension again. Also, from a circuit designer's perspective, operations can be decomposed and regrouped to achieve a more efficient data path, especially to memory. However, as our goal is to show that multi-core processors and ISE complement each other, it does not matter how efficient the ISE is per se.

4.4 Dual Core ISE Hardware Accelerated Version

We implemented a version of the FW that incorporates both multi-core and ISE techniques. We started with the dual core implementation described in Section 4.2. Then we applied the same design flow for the eMIPS Extensions used in Section 4.3 [7].

Like the sequential version (Figure 10), in the dual core



```
;; Hot block assembly for eMIPS CORE0
0660:  lw v0,0(t1)       Load dist[i][k], dist[k][j],
0664:  lw a0,0(a0)       and dist[i][j] from DDR2.
0668:  lw v1,0(a1)
066c:  addu a0,a0,v0     Compare dist[i][j] and
0670:  sltu v1,a0,v1     dist[i][k] + dist[k][j].
0674:  lui v0,0x8008
0678:  beqz v1,688
067c:  addu v0,t0,v0
0680:  sw a0,0(a1)       Update dist and pred
0684:  sw s0,0(v0)       matrixes with new values.
0688:  sll v0,a3,0x1
068c:  sll a0,a3,0x5
0690:  subu a0,a0,v0
0694:  addu v0,a2,s1
0698:  lui v1,0x8010
069c:  addu a1,a0,s0
06a0:  sll v0,v0,0x2     Increase loop index, jump
06a4:  addu a0,a0,a2     back if boundary has not
06a8:  sll a1,a1,0x2     been met.
06ac:  sll t0,a0,0x2
06b0:  addu t1,v0,v1
06b4:  slt v0,a2,a3
06b8:  addu a0,a1,v1
06bc:  addiu a2,a2,1
06c0:  bnez v0,660
06c4:  addu a1,t0,v1
;; Hot block assembly for eMIPS CORE1
03cc:  lw a0,0(a3)       Load dist[i][k], dist[k][j],
03d0:  lw v0,0(t1)       and dist[i][j] from DDR2.
03d4:  lw v1,0(a1)
03d8:  addiu a2,a2,1
03dc:  addu a0,a0,v0     Compare dist[i][j] and
03e0:  sltu v1,a0,v1     dist[i][k] + dist[k][j].
03e4:  lui v0,0xfff8
03e8:  slti t2,a2,30     Increase loop index,
03ec:  addiu a3,a3,4     compare with the boundary.
03f0:  beqz v1,400
03f4:  addu v0,a1,v0
03f8:  sw a0,0(a1)       Update dist and pred
03fc:  sw s1,0(v0)       matrixes with new values.
0400:  bnez t2,3cc       Jump back if boundary
0404:  addiu a1,a1,4     has not been met.
```

Figure 16. Hot Blocks for Parallel Floyd-Warshall in MIPS Assembly.

version (Figure 12) most of the application execution occurs within the inner loop of the second half of the application comparing new and old distances and updating the matrix if the new distance is better. Except in the case of the dual core version we have two instances of this loop now acting on different parts of the matrix in parallel. This means that we will have potentially two different hot blocks, one for each core. Since both versions of the inner loop are performing the same task on both cores, we would expect them to be similar. However, the difference in the boundary conditions of the two versions results in significant differences as the compiler attempts to optimize the code at the machine level.

The difference is reflected in the profiling results of the dual core implementation of the FW. The assembly representation of the hot blocks for CORE0 and CORE1 are given in Figure 16. We observe that while the dominant function of these basic blocks is the same operation of load, add, compare, conditional store and branch, CORE0 has significantly more instructions than CORE1 and both are longer than the sequential version.

Both the Extensions based on these hot blocks function similarly to the one implemented for the sequential version. After the Extension is activated by the Extension Instruction, the input registers are fetched from the general purpose register file. The register assignments differ between the sequential and these parallel versions of the hot block but they represent the same symbols of the pointers to *dist*[i][k], *dist*[k][j], *dist*[i][j], and *pred*[i][j] and loop counters *j* and *k*. Both Extensions add *dist*[i][k] and *dist*[k][j] and compare it to *dist*[i][j]. If the sum is less, *dist*[i][j] is updated with the sum and *pred*[i][j] is updated with the value of *k*. Finally, the Extension cleans up state by writing registers back to the general purpose register file and resuming pipeline execution.

In both cases, changes to the inner loop boundary conditions given in Figure 12 (Line 3 and Line 11) resulted in additional instructions for calculating the more complex boundaries. After careful inspection of the assembly of both hot blocks we find that the boundary conditions that partitioned the NxN matrix into two triangular pieces changed how the compiler traversed through the addresses of the matrix elements through the pointers.

In the hot blocks, pointers to *dist*[i][k], *dist*[k][j], *dist*[i][j], and *pred*[i][j] are given as inputs. In the sequential version (Figure 10) each of these is incremented by four except for *dist*[i][k] which is constant in this scope. In the dual core (Figure 12) version the simple increment by four is replaced in the CORE0 by a set of shifts and adds to constant base for each pointer. The hot block for CORE1 still increments by four for most of the pointers except for the *pred*[i][j] which is derived by adding the pointer to *dist*[i][j] to a constant.

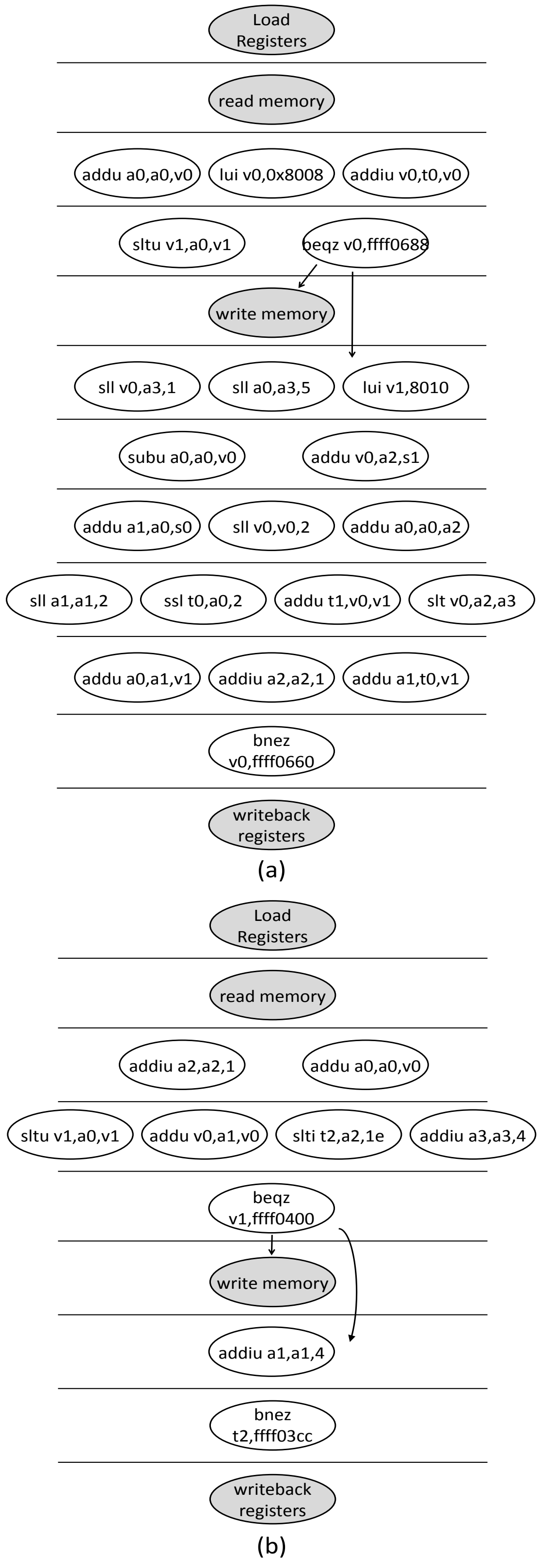


Figure 17. ISEs for Parallel Floyd-Warshall: (a) ISE for CORE0 and (b) ISE for CORE1.

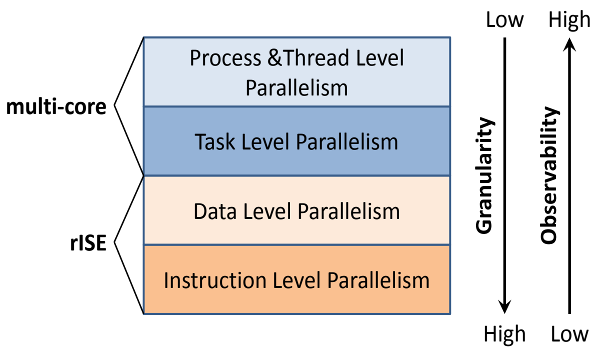


Figure 18. Different Levels of Parallelism.

These low level compiler 'optimizations' increases the number of instructions of the hot blocks by twelve and one for CORE0 and CORE1 respectively. The workload of the CORE0 is significantly increased over that of CORE1 and they are not as balanced as they initially appeared from the high level software representation. A flow chart of the FSMs for each of the parallel ISEs is given in Figure 17. They operate similarly to Figure 15 and their discussion is therefore omitted for brevity.

5. Discussion

In the previous section, we presented the design strategies and resulting implementations for four versions of the FW, three with different performance enhancing techniques. The last implementation combined the multi-core and ISE techniques of the other two. Both these techniques accelerate application performance by exploiting parallelism. However, each exploits different types of parallelism and in different ways. These techniques are not mutually exclusive where one provides all the benefits while the other adds little. To understand the synergy of these techniques it is important to discuss where the advantages of each are found. We illustrate this through our example of the FW and show how the combination of these techniques provides opportunities not available to either alone.

5.1 A System Perspective

From a system perspective, multi-core based parallel programming and ISE are two orthogonal methods that do not contradict each other. As we hope to have proven in this paper, their acceleration contributions can be added up to achieve higher performance. Moreover, although both of these methods deliver acceleration by exploiting parallelism, they are at different levels of granularity. This means that there is no concern that one of them handles all the parallelism while the other has nothing left to do.

In general, there are 4 levels of parallelism [28]: Instruction Level Parallelism (ILP), Data Level Parallelism (DLP), Task Level Parallelism (TLP), and Process and Thread Level Parallelism (PTLP). We observe that the types of parallelism suitable for ISE and multi-core vary along the parameters of granularity and observability. Multi-core based parallel programming works well when parallelism is coarse grained and has high observability. By comparison, ISE works well when parallelism is fine grained and observability is low because it resides underneath the software implementation. Figure 18 illustrates the overall picture of different levels of parallelism.

ILP has the lowest level of granularity. It is achieved by operating several different instructions in the same clock cycle. A critical feature of ILP is that it is not visible in the source code. Therefore, ILP is out of the programmers' control. We already showed that this 'out of control' characteristic also brings practical problems to the higher-level parallelism in Section 4.4. ISE is the perfect solution to exploit ILP. Also, ISE brings low-level control back to developers. We can use it to solve the practical problems we encounter.

DLP means that several different data elements can be processed in parallel. This level of parallelism is suitable to be exploited by ISE. For example, we can unroll the inner most loop of FW and process several matrix elements in parallel with the ISE.

TLP is achieved by processing several different blocks of codes, e.g. loops, in parallel. Parallelism at this level is too high to be exploited by ISE. Instead, multi-core becomes a more suitable solution. Consider the parallel FW case. We divide the middle loop in Figure 10 into 2 parallel smaller loops in Figure 12 and assign them to different eMIPS cores. In this way, we process two instances of a loop at the same time using two cores.

PTLP is even higher than TLP. It is the most prevalent parallelism we can see in parallel computing systems. Here, multi-core is the best solution.

5.2 A Multi-Core Perspective

By adding different hardware instruction set extensions to different eMIPS cores, we gradually change the homogeneous multi-core system into a heterogeneous one. The heterogeneous system gives developers more options and control for task partitioning among different cores. As a result, it is less likely to see some cores finish their jobs earlier and sit idle waiting for the others. This makes it easier to achieve balanced task partitioning, and therefore, it increases the efficiency of the multi-core system.

While the above benefit from ISE to multi-core is easy to see, there are also other benefits that are more subtle. In the rest of this section, we propose our solution based on ISE to remedy the assembly level imbalance problem mentioned in Section 4.4.

As discussed in Section 4.2, the task loads assigned to different eMIPS cores appear quite balanced. However, in Section 4.4 when we compile the program for each core with GCC (O2), we obtained the assembly codes for the dual-core system, shown in Figure 16. Although the C programs for the sequential FW and both CORE0's and CORE1's parallel FW all look very similar, the resulting assembly code shows significant differences. Specifically, by only changing the bottom or upper limit of the inner most loop's index, we find large differences in the assembly code. The number of instructions in CORE0's parallel FW implementation is much larger than that of the sequential FW. This means that although the number of iterations are reduced for each core in the dual-core system, the length of every iteration for CORE0 is increased. Thus, the overall acceleration will not be very high.

The increased complexity of CORE0's parallel FW is caused by the changes to the inner loop boundary conditions used to partition the NxN matrix into two triangle pieces. As discussed in Section 4.4, this changes the way the compiler updated the pointers to the matrix elements processed by the loop. This is because of the inefficiency of compilers when compiling irregular nested loops (the innermost loop's boundary is changing). This problem is usually out of the programmers' control.

In order to balance the task partitioning of different eMIPS cores, we accelerate the operations in CORE0 and CORE1 with ISEs, which are illustrated in Figure 17. The ISE for CORE0 only has 3 more states than CORE1's ISE. This means that the difference in execution time is only 3 clock cycles between the two cores. Compared with the previous difference (11 instructions), the task loads are much more balanced. In such a way, developers regain control over low-level issues again. This grants developers more power to achieve balanced multi-core systems. Similarly, for other applications that cannot be partitioned in a balanced way in the C program, ISE can also be used to re-balance the execution times of the various parts.

5.3 An ISE Perspective

Due to the low-level features of the ISE, it is difficult to achieve a global optimum in terms of speed and area cost, if we only look at one single ISE. Fortunately, high-level parallel programming on multi-core systems gives us a global view of an application. Thus, it helps to guide the design of ISE to a global optimum.

In Section 5.2, we showed an example where ISE could be used to benefit multi-core systems. With the same example, we can also find that the multi-core system makes ISE more efficient. As we mentioned, the difference between the two ISEs is 3 clock cycles. This means that CORE1 still has a lighter task load. In such a case, we can lower the speed requirement for CORE1. Our solution is to add 2 additional states to CORE1's ISE and reduce the number of instructions performed in one state. Figure 19 shows the improved operation procedure. Consequently, the difference in the number of states in CORE0 and CORE1 is reduced to 1. The overall performance remains the same. However, CORE1's ISE takes less area. Previously, there were at most four operations processed in parallel in one clock cycle. The hardware resources needed to accomplish that are two comparators and two adders. After the optimization, the ISE only requires at most one comparison and one addition in each state. The hardware resources needed are reduced to one comparator and one adder. Experimental results in the next section will show that it takes approximately 6% less area. If the reused resources were more complicated, for example multipliers, or consisted of more computational elements, the area reduction would be higher.

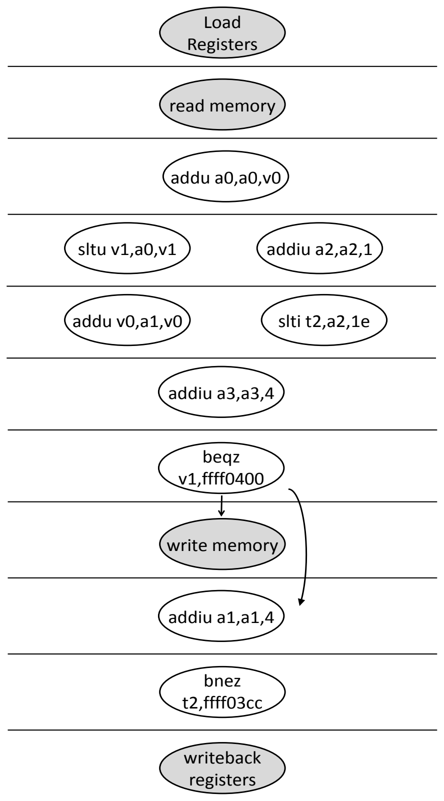


Figure 19. More Efficient ISE for CORE1.

The above analysis tells us that parallel programming on multi-core gives developers a high-level view of a problem. Therefore, they can discover more globally efficient ISE implementations solutions that are locally sub-optimal in one dimension, but more efficient in another dimension.

6. Experimental Results

In order to demonstrate our analysis in the last section, we tested four versions of Floyd-Warshall algorithm: *Seq*, *SeqExt*, *Para*, and *ParaExt*. The relationships between these different versions is summarized in Table 1.

Table 1. Summary of Different Versions of FW

	Sequential	*Parallel*
No extension	Seq	Para
With extensions	SeqExt	ParaExt

Seq is our baseline implementation of Figure 10. It is a single threaded program that sequentially updates each entry in the matrix. Para is our dual core implementation splitting the matrix diagonally allowing each core to update half the matrix (Figure 12).

We profiled both of these programs and identified that the basic block with the highest execution frequency is the most inner loop of the algorithm. The profile revealed that the assembly code of the three versions of the innermost loop were similar but not exactly the same. This required us to implement three different extension designs.

The *SeqExt* version implements *Seq* accelerated by the extension hardware (shown in Section 4.3). The extension instruction is inserted into Seq's binary using our patching tool. Similarly, *ParaExt* is *Para* implemented with two different extensions loaded into each of the two eMIPS cores (the combined method discussed in Section 4.4 and 5). The binaries of each core are also patched with the appropriate extension instructions.

The experiments were performed on a Virtex 5 (xc5vlx110t-2ff1136) FPGA, where our dual-core eMIPS design including peripheral and memory support were implemented. For each experiment, the algorithms were run on one or both cores with the instructions stored in the local BlockRAM of the host core and the graph matrix data is stored in the DDR2 RAM shared by both cores. The algorithms were applied to a random weighted directed graph of 30 vertices. Measurements were taken using the local timer, running at 10 MHz.

Table 2, presents the execution time for all versions.

Table 2. Execution Time of Different Versions of FW

	Seq	*SeqExt*	*Para*	*ParaExt*
Execution Time (ms)	25.85916	12.10826	22.72854	7.00032

To compare the acceleration of the various techniques, the speed ups are presented in Table 3. We can see that parallel programming without extensions only gives us a speedup of 1.138. Also, we find that the hardware instruction set extension in *SeqExt* gives a 2.136 speedup over the version Seq. After combining parallel programming and ISE together, we find *ParaExt* has a speedup of 3.7 over *Seq* and 1.73 over *SeqExt*.

Table 3. Speedup of Different Versions of FW

	Seq	*Para*	*SeqExt*
Seq	1	--	--
Para	1.138	1	--
SeqExt	2.136	1.88	1
ParaExt	3.7	3.25	1.73

Table 4. Area Cost of the Original ISE and the more efficient ISE for CORE1 in *ParaExt*

	#adders	*#comparators*	*#registers*	*#LUTs*
Original	11	2	676	1148
Improved	10	2	677	1080

We also implemented the more efficient ISE for the *ParaExt* version of CORE1 as shown in Figure 19. Experimental results show that *ParaExt*'s execution time with the more efficient ISE is 7.0178ms. The performance is maintained. After synthesis using Xilinx ISE 10.1, the area costs of the original ISE and the improved ISE are different. The resource requirements of the two implementations are listed in

Table 4. The numbers of adders and comparators are obtained from the 'Advanced HDL Synthesis Report'. The numbers of registers and LUTs are obtained from the 'Device utilization summary'.

7. Analysis

The acceleration of *Para* over *Seq* was only 1.138. The efficiency per core is only 0.57 – surely not satisfying. This shows that although the task partitioning in *Para* divides in half the number of matrix elements that each core processes by 2, the overall performance does not improve much because CORE0 in the *Para* version executes almost twice the number of instructions as in the *Seq* version for every matrix element (Figure 16). We experimentally rule out the possibility that this lukewarm performance was mainly due to memory contentions: that is, when both cores attempt to access the shared memory at the same time, one core will be delayed until the other finishes. From the RTL simulation, we noticed that most memory access operations from the two cores were interleaved. Even when the contention happened, the delay of one core was small compared to the overall memory latency. This observation shows that memory contentions did not reduce *Para*'s performance by a large extent. From another perspective, in contrast with *Para*, two cores in *ParaExt* need to finish the same sets of memory access operations within a much shorter time. Thus, *ParaExt* has a more severe memory contention problem than *Para*. But this did not result in a low speedup of *ParaExt* over *SeqExt* (1.73). Therefore, for the second time, we have shown that memory contentions are not the major problem for the *Para* version in this case.

More importantly, the speedup of *ParaExt* over *SeqExt* is 1.73. This indicates that, after adding ISEs, the dual-core system's efficiency per core increases from 0.57 to 0.87. This improvement comes from the more balanced execution time between the two cores. Since the gap in the work load between the two cores is significant reduced using the ISEs, CORE1 no longer has to wait large amounts of time between iterations for CORE0 to complete its section of the matrix. Therefore, we demonstrate that ISEs increase multi-core systems' efficiency by balancing the execution time of different cores.

According to the area costs in

Table 4, the more efficient ISE uses one fewer adder than the original ISE. Accordingly, the more efficient ISE takes 68 fewer LUTs. The synthesis tool did not reuse the comparator. The reason for this is that compared to the cost of reusing a comparator (e.g. multiplexers), this optimization is not worthwhile. We also notice that one more register is needed by the more efficient ISE. The detailed synthesis report shows that two more registers are used for two additional states (speed1 encoding). However, the original ISE has one more register that is duplicated. Thus, the improved ISE costs one more register. Overall, the more efficient ISE reduces the LUT cost by 6% while maintaining the same performance. Therefore, it appears that multi-core techniques can also be used to make ISE more efficient.

The speedups of *Para* and *SeqExt* over *Seq* are 1.138, and 2.136 respectively. After combining multi-core processors and ISEs together, we obtain a speedup of 3.7 from *ParaExt*. The acceleration is higher than either one used in isolation. Moreover, the speedup of *ParaExt* is even higher than the product of Para and *SeqExt* alone (3.7>1.138*2.136). This demonstrates the higher efficiency obtained by the combined method. This shows that, by exploiting different levels of parallelism, multi-core processors and ISEs add their accelerations together and realize a high-performance system in an efficient way.

Note that a number of research efforts have investigated high-performance FW implementations on FPGAs [29] [30] and GPUs [31] [32]. We do not compare our case study with them. This paper proposes a highly flexible architecture, not a specific dedicated design. The performance of the new architecture depends on the performance of the specific multi-core processors and the ISEs. Considering the case study only uses two soft eMIPS cores with two unoptimized ISEs, it is not fair to compare it with the previous optimized designs. What the case study really shows is that the combination of multi-core parallel programming and ISEs produces overall better performance with higher efficiency than solutions based on only one of the two.

8. Conclusion

In this paper, we proposed multi-core processors combined with reconfigurable instruction set extensions as a high performance architecture. Both analysis and experimental results show that these two acceleration methods not only exploit different levels of parallelism, but also benefit each other. Therefore, we consider the combination as a promising solution for high performance systems.

REFERENCES

[1] T. M. Pinkston and J. Shin, Trends Toward On-Chip Networked Microsystems, *Int'l J. High Performance Computing and Networking*, vol. 3, no. 1, Dec. 2005, pp. 3-18.

[2] NVIDIA, NVIDIA GeForce 8800 GPU Architecture Overview, http://www.nvidia.com/object/IO_37100.html.

[3] Intel, Intel Xeon Processor 5500 Series, http://download.intel.com/products/processor/xeon/dc55kprodbrief.pdf.

[4] AMD, Six-Core AMD Opteron Processor Product Brief, http://www.amd.com/us/products/server/six-core-opteron/pages/six-core-opteron-product-brief.aspx.

[5] G. M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conerence Proceedings, vol. 30 (Atlantic City. N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

[6] H. Lu, A. Forin, The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs, MSR-TR-2007-99, Microsoft Research, WA, Aug. 2007.

[7] R. N. Pittman, N. L. Lynch, A. Forin, eMIPS, A Dynamically Extensible Processor, MSR-TR-2006-143, Microsoft Research, WA, Oct. 2006.

[8] U. Bondhuqula, A. Devulapalli, J. Fernando, P. Wyckoff, P. Sadayappan, Parallel FPGA-based all-pairs shortest-paths in a directed graph, Parallel and Distributed Processing Symposium, 2006, IPDPS 2006, pp. 25-29, April 2006.

[9] K. Öner, L.A. Barroso, S. Iman, J. Jeong, K. Ramamurthy, and M. Dubois, "The Design of RPM: An FPGA-based Multiprocessor Emulator,"

International Symposium on Field-Programmable Gate Arrays (FPGA95), pp. 60-66, 1995.

[10] J.D. Davis, S.E. Richardson, C. Charitsis, and K. Olukotun, "A Chip Prototyping Substrate: The Flexible Architecture for Simulation and Testing (FAST)," ACM SIGARCH Computer Architecture News, Vol. 33, No. 4, pp. 34-43, 2005.

[11] J. Wawrzynek, D. Patterson, M. Oskin, Shih-Lien Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanović, "RAMP: Research Accelerator for Multiple Processors," IEEE Micro, Vol. 27, No. 2, pp.46-57, 2007.

[12] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: A Chip-Multiprocessor with Transactional Memory Support," Design, Automation and Test in Europe (DATE2007), pp. 3-8, 2007.

[13] C. Chang, J. Wawrzynek, and R.W. Brodersen, "BEE2: A High-End Reconfigurable Computing System," IEEE Design & Test, vol. 22, no. 2, pp. 114-125, 2005.

[14] D. Lee, "OpenSPARC – A Scalable Chip Multi-Threading Design," International Conference on VLSI Design, (VLSID2008), pp. 16-16, 2008.

[15] BEE3: http://research.microsoft.com/projects/BEE3/

[16] D. Vahia and P. Hartke, "OpenSPARC T1 on Xilinx FPGAs – Updates," 2007, DOI= http://ramp.eecs.berkeley.edu/Publications/OpenSparc%20%286-14-2007%29.pdf

[17] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications," IEEE Journal of Solid-State Circuits, Vol. 38, No. 11, pp. 1876-1886, 2003.

[18] M. Epalza, P. Ienne, and D. Mlynek, "Adding Limited Reconfigurability to Superscalar Processors," Parallel Architecture and Compilation Techniques (PACT2004), pp. 53-62, 2004.

[19] Tensilica, Inc. "Xtensa 7 Product Brief," http://www.tensilica.com/uploads/pdf/xtensa_7.pdf.

[20] MIPS Technologies, Inc. "Pro Series Processor Cores," Application Notes, http://www.mips.com/media/files/Pro%5FSeries.pdf.

[21] ARC, Inc. "ARC 600 Configurable Core Family," http://www.arc.com/evaluations/15_ARC_600_Family.pdf.

[22] Stretch Inc. "S6000 Family", Product Brief, http://www.stretchinc.com/_files/S6000.pdf.

[23] K. Atasu, G. Dűndar, and C. Özturan, "An Integer Linear Programming Approach for Identifying Instruction-Set Extensions," International conference on Hardware/Software Codesign and System Synthesis, pp. 172-177, 2005.

[24] K. Meier and A. Forin, "MIPS-to-Verilog, Hardware Compilation for the eMIPS Processors" Microsoft Research Technical Report MSR-TR-2007-128, 2007, http://research.microsoft.com/en-us/projects/emips/tr-2007-128.pdf.

[25] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific Instruction Generation for Configurable Processor Architectures," International Symposium on Field Programmable Gate Arrays (FPGA2004), pp. 183-189, 2004.

[26] R.N. Pittman, N.L. Lynch, and A. Forin, "eMIPS, A Dynamically Extensible Processor," Microsoft Research Technical Report MSR-TR-2006-143, 2006, http://research.microsoft.com/en-us/projects/emips/emipsreport1.pdf.

[27] R.W. Floyd, "Algorithm 97: Shortest Path," Communications of the ACM, vol. 5, page 345, 1962.

[28] S.A. Ostadzadeh and K. Bertels, "Parallelism Utilization in Embedded Reconfigurable Computing Systems: A Survey of Recent Trends," The Journal of VLSI Signal Processing, Vol. 28, No. 1-2, pp 7-27, Springer, 2004.

[29] U. Bondhuqula, A. Devulapalli, J. Fernando, P. Wyckoff, P. Sadayappan, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," Parallel and Distributed Processing Symposium (IPDPS 2006), pp. 25-29, 2006.

[30] E.I. Milovanvić, I.Ž. Milovanović, M.P. Bekakos, I.N. Tselepis, "Computing All-Pairs Shortest Paths on A Linear Systolic Array and Hardware Realization o a Reprogrammable FPGA Platform," The Journal of Supercomputing, Vol. 40, No. 1, pp. 49-66, 2007.

[31] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, "Scan Primitives for GPU Computing," 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 97-106, 2007.

[32] G.J. Katz and J.T. Kider Jr, "All-Pairs Shortest-Paths for Larger Graphs on the GPU," 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 47-55, 2008.

[33] Sekar, A., Forin, A. Automatic Generation of Interrupt-Aware Hardware Accelerators with the M2V Compiler., MSR-TR-2008-110, Microsoft Research, WA, August 2008.