Accelerating Coupled Applications through Register Level Communication between Processing Elements*

Chengmo Yang and Alex Orailoglu
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
{c5yang, alex}@cs.ucsd.edu

ABSTRACT

Early SoCs have boosted parallelism exploitation for a limited number of embedded system applications that can be easily decomposed into multiple independent parts, thus enabling their facile execution on multiple Processing Elements (PEs) in parallel. However, as the computation complexity of the applications increases, the lockstep execution model is increasingly being questioned due to architectural considerations, such as cache misses and branch mispredictions. As a result, it becomes imperative to provide organizational models that are capable of providing flexible execution, adapting to the vicissitudes of the run time environment. In this paper, we provide a new architectural organization framework, capable of extracting as much as possible application information through static analysis, as well as dynamic adjustment of the operations to suit the possible unpredictability in run-time events, all within a negligibly small amount of hardware overhead. Since such flexible execution models have previously had to resort to increasing amounts of communication coherence checking, we additionally propose a novel collaborative communication framework that can be customized on an application basis to reduce the communication overhead between PEs. It is shown that the supporting algorithms are provably correct; furthermore, mathematical analysis confirms the superiority of the proposed techniques.

1. INTRODUCTION

As IC technology advances, the number of transistors available on a single chip attains dramatic scales of integration, unfortunately though matched by commensurately dramatic increases in the complexity of current embedded applications. As a result, the challenge imposed on current embedded system designers consists of designing a well organized system that can efficiently utilize the billions of transistors available as well as effectively exploit the potential parallelism of the complex application as much as possible.

In the traditional computer architecture domain, nearly all personal computer and workstation processors boost performance in the form of superscalar or very long instruction word (VLIW) architectures. However, neither of these approaches fits perfectly the requirements of the embedded system domain. For superscalars, the hardware implementation of the aggressive dynamic scheduling algorithm [1] is rather complex and expensive, resulting in ever increasing power consumption as dynamic issue width grows [2]. As for VLIWs, the paramount problem consists of the efficiency of static scheduling [3] to schedule independent instructions to fill in the ever increased number of issue slots per cycle. Furthermore, since VLIW architectures only contain a single program counter and branch mechanism, if even a single operation is unable to execute, the complete VLIW instruction has to be stalled, with significant implications on performance degradation.

The inability of both the traditional superscalar and the VLIW approach to effectively respond to the challenges of current high-end embedded system applications motivates designers to exploit parallelism in the form of a decentralized microarchitecture, such as Chip Multiprocessors (CMPs). The corresponding hardware functionality model is System-on-Chips (SoC), which provides multiple Processing Elements (PEs) on a single chip. The embedded applications to be executed on a SoC are decomposed into independent parts in order to be executed on different PEs. The synchronizations and communications between PEs are typically performed through an on-chip network or shared memory. However, both communication schemes have a quite expensive latency of hundreds of clock cycles, constituting the bottleneck of this multiprocessor execution model. For coupled embedded applications, decomposing them into large independent parts is almost infeasible, motivating the construction of an efficient On-chip Communication Architecture (OCA). Furthermore, the efficiency of this lockstep execution model is increasingly being questioned due to the ever increased number of unpredictable run-time events, such as cache misses in the data subsystem, and branch execution in the control subsystem. As a result, it becomes imperative to provide organizational models that are capable of providing flexible execution, by adapting to the vicissitudes of the run time environment. Such flexible execution models have typically necessitated increasing amounts of communication coherence checking, putting in turn emphasis on the construction of a well organized OCA for embedded systems, capable of providing cheap yet efficient communication between PEs during execution.

To accelerate coupled embedded applications, we propose a register level communication scheme in this paper. As an improvement to the traditional SoC model, our framework enables different parts of the application to communicate during execution, even though they are decomposed to be executed on different PEs. Our framework relies on the compiler to automatically parallelize the applications into multiple parts in order to be executed onto different PEs, simplifying the work of programmers. The cheap yet efficient com-

*The work of the first author is supported in part by NSF Grant 0082325.

munication scheme is provided by allowing two adjacent PEs to share several *communication registers*. The insertion and scheduling of the necessary communications in the code are performed statically by the compiler, simplifying the work of operating systems. Although the techniques we propose can be applied to personal computers as well as workstations and provide a departure point for future research in that domain, they can be thought of as customizable platforms for embedded systems, which execute well-defined applications whose information can be exploited tremendously by the compiler.

The main challenge of implementing register level communications consists of the prevention of potential semantic violations. Suppose a write operation to a particular communication register is followed by a read operation to the same register emanating from a different PE. The semantically-correct dependence between these two instructions constitutes a *Read After Write (RAW) dependence*, that is, the second instruction needs the value produced by the first instruction. However, because of unpredictable run-time events, the first PE may be delayed, causing the write operation to access that register later than the read operation. With no static information, the dynamic check mechanism may misinterpret this dependence as a *Write After Read (WAR) dependence*, resulting in the read operation obtaining an incorrect value. This potential semantic violation constitutes a **synchronization violation**. In our architecture, a novel ***reference coloring algorithm*** is proposed to explicitly encode dependence information statically for each communication. While the multiplicity of interdependent communications and the arbitrary order of such potential communication relationships may force consideration of significant hardware overhead, in precluding such semantic violations, we propose an approach wherein the statically encoded dependence information is to be checked dynamically with only **2-bit** hardware overhead per communication register. The static encoding and dynamic checking together prevent synchronization violations by forcing all the instructions that access a specific communication register to be executed in order, even if they are executed on different PEs.

Two other questions are also of critical importance to the proposed architecture. A design question concerns the restriction to adjacent register level communications between PEs. The restriction is imposed by the need to localize such register-level communications to simplify both the static scheduling algorithm and the dynamic checking hardware. A performance question concerns how to make each PE execute instructions smoothly without spending too many cycles awaiting data transfer from another PE. Standard extensions of compiler static scheduling techniques can be utilized for this purpose.

The remainder of this paper is organized as follows. Section 2 reviews previous communication and synchronization techniques. Section 3 discusses in detail the architectural and the compiler framework on which our new communication technique is to be applied. Section 4 presents our reference coloring scheme to be used to guarantee semantically correct communications, including the static encoding algorithm and the dynamic checking mechanism. Section 5 provides a mathematical evaluation framework that considers the two distinct cases of PEs being VLIW or superscalar processors. Section 6 summarizes this paper and suggests possible future research avenues in this domain.

2. PREVIOUS WORK

In the design of superscalar processors, the handling of *RAW* and *WAR* dependences is an old topic. The traditional dynamic scheduling approach, such as the well-known Tomasulo's algorithm [1], employs register renaming techniques to eliminate *WAR* and *RAW* dependences. This approach enables handling some cases when dependences are unknown at compile time (e.g., the dependences involve a memory reference). However, the hardware scheduler is restricted to selecting operations from a fixed-size window, and thus does not have the breadth of information available to it that a compiler would have at compile time. Enlarging the window size results in a longer average latency for all operations, and most importantly high hardware costs and power consumption for dependence checking and resource allocation.

In current embedded system domain, Ideally there exist three different approaches to exploit the potential parallelism provided by the SoC model, which has multiple Processing Elements (PEs) on a single chip [4]. The first approach is to execute multiple processes in parallel to increase throughput in a multiprogramming environment. It relies on a multiprocessor aware operating system to control the communication and synchronization between different processes, and to efficiently schedule multiple processes onto multiple PEs. For most embedded systems, complex operating systems are not provided, limiting the applicability of this approach. The second approach consists of executing multiple threads in parallel that come from a single application. While the scheduling of threads **within** each PE can be performed dynamically by hardware, an operating system is still necessitated to schedule threads **across** PEs [5]. In addition, this method relies on the programmer to explicitly parallelize the application into multiple threads, requiring a deep understanding of the nature of the parallelism in the application. The third approach automatically divides the sequential applications into multiple parts and executes them on different PEs. However, because of the significant communication overhead associated with the current SoC model, this automatic parallelization technology is only effective for applications with a quite limited number of dependences, such as scientific applications [6].

To improve the applicability of automatic parallelization, we propose a cheap yet efficient register level communication scheme. Previously some other register level communication schemes have also been proposed in the design of multiprocessor architectures, such as the *multiscalar processors* [7] and the *Multi-Level Computing Architecture* (MLCA) [8]. However, while in our scheme all the checking and handling of inter-PE dependences are distributed among PE themselves, in both multiscalar processors and MLCA, a centralized control processor is employed to schedule different tasks onto PEs as well as check and handle the data and control dependences between PEs. In addition, some constraints which may impact performance are imposed on either multiscalar processors or MLCA. For example, in a multiscalar processor architecture, the PEs are forced to be connected by a unidirectional ring-type network in order to simplify the checking and handling of dependences, imposing a sequential order among the PEs. In the MLCA, a *uniform register file* (URF) is employed to implement the communi-

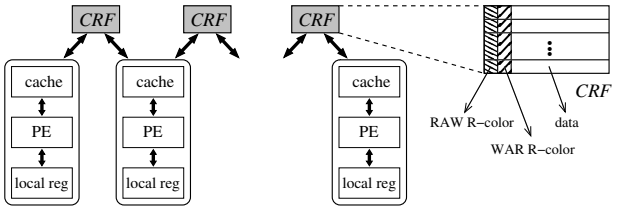

Figure 1: Architectural overview

cations between tasks. Each task needs to check the URF to ensure that all the data are ready before been executed, implying that the communication between tasks are limited to be performed either before or after execution.

Similar to the previous register level communication scheme, our framework also relies on the compiler to decompose an application into different parts. However, these parts can communicate with each other **during** execution by allowing two adjacent PEs to share several communication registers. In addition, our communication scheme employs a static scheduling to aggressively exploit application information, as well as a dynamic mechanism to adjust operations according to unpredictable dependences and run-time events, all to be accomplished within a negligibly small amount of hardware overhead.

3. THE PROPOSED FRAMEWORK

3.1 Architectural-level overview

As presented in Figure 1, multiprocessors composed of PEs, each PE able to issue multiple instructions, and all processors executing possibly asynchronously to each other with their instructions emanating from their own individual level-one instruction cache (ICache) can be envisioned as a possible framework on which our proposed technique is to be applied. Each PE is presumed to have its own local register file and maintains all state information required to function independently, thus enabling the PEs to execute completely unrelated tasks in parallel. PEs can furthermore collaborate to execute a single application, since dynamic communications and synchronizations between them are also enabled through a number of communication registers between adjacent PEs. The proposed framework is quite flexible, enabling the number of internal PEs to vary across a diverse span of architectures. A measure of the associated flexibility can be seen by noticing that PEs can either be *superscalar* or *VLIW* processors, or any combination thereof.

In general, dependences in the proposed architecture can be classified into two types, based on their locations. ***Local dependences*** occur when an instruction pair that forms a dependence is restricted to lie within the same PE. Since each PE is a small superscalar or VLIW processor, local dependences can be handled perfectly. ***Global dependences***, on the other hand, occur when the instructions of the dependent pair lie in distinct PEs. In this case, instead of a local register, a communication register is specified in the source or destination register field of the dependent instruction pair. During the compilation process, global dependences are signaled explicitly. To guarantee semantic correctness, the instructions with global dependences must access the corresponding communication register in order, even if the associated accesses are emanating from different PEs. The challenge as mentioned previously consists of forcing the in-order execution necessary for semantic correctness through without a significant amount of hardware overhead and performance degradation. In our framework, a novel reference coloring scheme is proposed to encode global dependence information necessary to ensure semantic correctness in arbitrary access contexts within only a **2-bit** overhead per communication reference. A dynamic checking mechanism is employed to check the dependence information during execution to prevent semantic violations.

Our architecture provides latency tolerance between PEs so that each PE can execute its instruction stream with its own timing between synchronous phases. If one PE stalls, the other PEs do not necessarily stall except in so far that their execution globally depends on the stalled PE. Another advantage of the proposed architecture is the distribution of local register files, which not only reduces the number of read and write operations needed to be performed simultaneously, but also enlarges the available register space through fully overlapping the register name space of each PE. Given that the hardware complexity and power consumption increase significantly as the size of register files grows, our architecture proves to be more economical for embedded systems than the architectures with centralized register files, such as superscalars and VLIWs.

3.2 Compiler-level overview

Effective implementations, capable of delivering improved performance at negligible hardware and power costs, typically necessitate a collaboration between architectural and compiler extensions. Compiler techniques contribute to the proposed approach by decomposing the application into various fragments, and specifying the necessary global dependences explicitly by using communication registers as source or destination registers for each dependent instruction pair during code generation. In addition, if internal PEs are VLIW processors, the compiler bears additional responsibility for packaging independent operations into a long instruction that can be executed on a PE directly. Since frequent communications between different PEs may cause performance degradation by introducing unnecessary stalls, the compiler also needs to minimize redundant global dependences so that a global dependence is presented only when and where it is absolutely necessary.

In order to minimize redundant global dependences, instruction pairs that have dependences should be scheduled onto the same PE as much as possible. More concretely, if operation j needs the result produced by operation i, scheduling j onto a different PE should be contemplated **only** in the case when the PE where operation i is scheduled is unavailable, a possible but infrequent occurrence stemming from operation i having dependents in excess of the capacity of the PE at that particular execution point. In this situation, non-critical dependent operations (operations that have few dependents or a short dependence chain) should be scheduled onto different processors. If operation j has to be scheduled onto a different PE, the PEs adjacent to the original PE need constitute the first choice, as extra *move* operations would be incurred to transfer the result produced by operation i, otherwise.

4. THE PROPOSED REFERENCE COLORING SCHEME

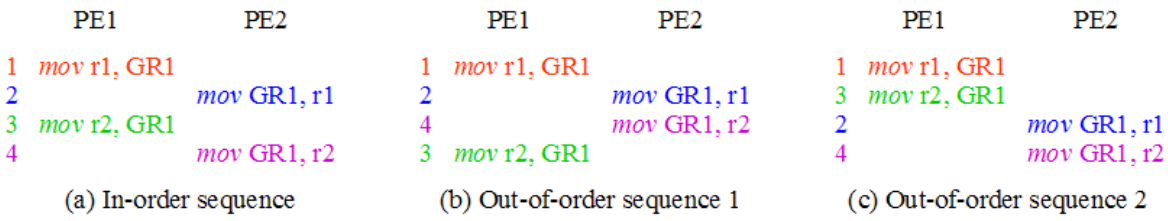

Figure 2: Out of order execution sequence

4.1 Ordering communication register accesses statically

The main challenge of implementing register level communications consists of the prevention of potential semantic violations, which are caused by unpredictable runtime events, such as branch misprediction and cache misses. This can be illustrated more clearly by considering in Figure 2 the following instruction sequence executed on two adjacent PEs.

In the example shown, there exist two global RAW dependences between instructions 1 and 2, as well as instructions 3 and 4, respectively. In addition, there also exists a global WAR dependence between instructions 2 and 3. These three dependences require all four instructions to be executed in order to guarantee semantic correctness (Figure 2 (a)). However, the possible unpredictability in runtime events may cause PE1 to be delayed, resulting in instruction 3 accessing GR1 later than instruction 4 (Figure 2 (b)). Similarly, PE2 can also be delayed, causing instruction 2 to access GR1 later than instruction 3 (Figure 2 (c)). In the former case, the original RAW dependence between instructions 3 and 4 is violated, while in the latter case, the original WAR dependence between instructions 2 and 3 is violated.

Guaranteeing semantic correctness necessitates that instruction 4 be stalled if it accesses GR1 before instruction 3. This can be achieved by forcing instruction 4 to wait on instruction 3, that is, in each usage of a value explicitly specifying its producer. Similarly, instruction 3 needs to be stalled if it accesses GR1 before instruction 2. This can be achieved by forcing each write operation to ensure the completion of previous read operations. One simple solution to prevent the violations of RAW dependence consists of including the address of the producer in each usage of a particular datum. However, the overhead of this solution is nontrivial, since the instruction address will cost at least 32-bit overhead.

As a comparison, the reference coloring algorithm we propose uses one **RAW-coloring field** and one **WAR coloring field** to handle the *RAW dependences* and *WAR dependences*, respectively. The *WAW dependences*, indicating that there exists no read operation between two adjacent write operations, are not considered, as they will presumably be eliminated by the compiler, since the value stored by the first write operation is useless. Similarly, if a write operation is followed by a read operation emanating from the same PE, the compiler will also reduce the number of communication register accesses by replacing the usage of that communication register with the PE's local register.

Handling *RAW dependences* requires distinguishing each write operation from the writes adjacent to it, and forcing each read operation to ensure that the corresponding write operation (the producer) has been performed. In our coloring algorithm, this is achieved by forcing the producer to set the **RAW-coloring field** to a particular *RAW color*, and forcing the corresponding consumers to assure the **RAW-coloring field** is correct. During the static compilation process, two *RAW colors* are used alternatively to make sure two adjacent write operations have different *RAW colors*, implying **one** bit is enough for the **RAW-coloring field**.

The handling of *WAR dependences* though imposes additional challenges over and above those encountered in the handling of *RAW dependences*. The underlying reason is that each consumer (read operation) has only one corresponding producer (write operation) to check in the case of RAW dependences, while each producer can have more than one consumer. As a result, in the case of *WAR dependences*, no write operation can be performed until all the previous read operations, that is, the consumers of a previous producer, have been executed. This implies that all the previous read operations should be assigned a distinct color in order for the write operation to perform checking, constituting a significant amount of overhead.

To simplify the case of *WAR dependences*, in our framework, the content of a communication register is copied to each PE's local register space for them to use during execution. The communication registers are only employed to perform data transfers, implying **one** consumer is generated to get the data of each producer. As a result, two *WAR colors* are enough to differentiate the read operations that need the results of adjacent write operations, implying **one** bit is enough for the **WAR-coloring field**. Furthermore, only utilizing communication registers for data transfer also reduces the necessary lifetime of values stored in communication registers, which in turn enables more reuse of the communication registers. As a result, the size of the communication register file can be reduced, which in turn reduces power consumption.

A pseudocode for a slight extension to the static compiler in order to incorporate the necessary updates for generating the suggested coloring bits can be undertaken as described below.

```
For each particular communication register {
  Order all the references emanating from two adj-
    acent PEs to that communication register.
  If two sequential references are read operations,
    or are write operations emanating from the same
    PE /* they constitute redundant references that
    need to be pruned */
  then prune them
  else {
```

Usage	Busy	RAW-coloring	WAR-coloring
Source	check before execution	check before execution	set during execution
Destination	check before execution set during execution reset after execution	set during execution	check before execution

Table 1: Check, set and reset of the three bits in communication registers

```
    Assign one RAW color and one WAR color to each
      write operation, with the property that two
      adjacent write operations have distinct RAW
      and WAR colors.
    Color each read operation using the same RAW
      color as the write operation immediately pro-
      ceeding it, while using the same WAR color as
      the write operation just following it.
  }
}
```

In this algorithm a total of two RAW colors and two WAR colors suffices, as it can be easily inferred that the overhead associated with each reference to a communication register consists of one **RAW-coloring bit** and one **WAR-coloring bit**. As long as each PE refrains from reordering the instructions with global dependences on its own, the coloring algorithm will prevent the potential synchronization violation by forcing a static explicitly specified dependence chain for all the references emanating from adjacent PEs to a specific communication register[1].

In the case when PEs are VLIW processors, a dynamic reordering is not performed. Therefore, semantic correctness can be guaranteed directly. If PEs are superscalar processors, the reordering of the instructions with global dependences can be prevented by disabling register renaming of the communication registers. In this case one active access to a communication register will block all the following accesses emanating from that PE.

4.2 Implementing synchronization through dynamic checking

Three extra bits are used per communication register to implement the dynamic checking mechanism. The first consists of a ***busy*** bit, which indicates whether an active operation is to write to that register. The checking, setting, and resetting of the ***busy*** bit in the case of a communication register occurring as a destination register guarantees that no other active instruction is interested in writing into that communication register; the occurrence of *WAW* hazards is thus prevented. The other two are the ***RAW-coloring*** bit and the ***WAR-coloring*** bit, which constitute the hardware support for the reference coloring algorithm to force the instructions which have global dependences to be executed in order. In instructions which have global dependences, the communication registers will occur either as source registers or destination registers or both. The following two cases delineate the dynamic checking, setting, and resetting of these three fields during dynamic execution as shown in Table 1.

- The source register is a communication register: During the decode stage, if the *busy* bit of the communication register is 0 **and** the *RAW-coloring* bit has the same color as the RAW color encoded in the instruction's source register, the instruction can proceed to be executed. Otherwise, the instruction needs to be stalled. The execution of this instruction will set the *WAR-coloring* bit of the communication register to the WAR color encoded in its source register.

- The destination register is a communication register: During the decode stage, if the *busy* bit of the communication register is 0 and the *WAR-coloring* bit has the same color as the WAR color encoded in the instruction's destination register, the instruction can proceed to be executed. Otherwise the instruction needs to be stalled. Before entering the execution stage this instruction will set the *busy* bit to 1, and set the *RAW-coloring* bit of the communication register to the RAW color encoded in its destination register. Once the execution is completed, the *busy* bit is reset to 0.

The reference coloring scheme discussed in the previous section explicitly encodes the dependence information between adjacent references to a particular communication register. This information gets checked during run time and execution flow on one PE is blocked if necessary to prevent a potential synchronization violation.

To illustrate the reference coloring algorithm and the dynamic checking mechanism more concretely, let us consider the example presented in Figure 2 once more. Assume that the two *RAW colors* used by the reference coloring algorithm are **red** and **green**, while the two *WAR colors* are **purple** and **blue**. During the compilation process, the RAW color and the WAR color of each instruction are specified as presented[2] in Table 2.

An illustrative case can be examined if we assume a dynamic access order of (1, 2, 4, 3) to the communication register GR1, as shown in Figure 2(b). The operation performed and the status of the *RAW-coloring* and *WAR-coloring* fields are presented[3] in Table 3. As can be seen, although instructions try to access the communication register GR1 in the semantically incorrect order of (1, 2, 4, 3), instructions are

[1]In case of further interest, the reader can peruse the rationale for this provided in the Appendix.

[2]As instruction 1 in this example is assumed to be the starting instruction, its corresponding *WAR color* is immaterial and no need exists for checking it; it is shown blank in the example.

[3]Please note that Step 7 is a replay of step 4 since the blocking condition has been cleared through the execution of steps 5 and 6.

Instruction	1	2	3	4
RAW color	red	red	green	green
WAR color		blue	blue	purple

Table 2: The result of coloring algorithm performed on the instruction sequence of Figure 2

Steps	Instruction	Operation	RAW-coloring bit		RAW-coloring bit	
			Before	After	Before	After
1	1 (write from PE1)	Set the RAW-coloring field to **red**		Red		
2	2 (read from PE2)	Block until the RAW-coloring field is **red**	Red	Red		
3	2	Set the WAR-coloring field to **blue**	Red	Red		Blue
4	4 (read from PE2)	Block until the RAW-coloring field is **green**	Red	Red	Blue	Blue
5	3 (write from PE1)	Block until the WAR-coloring field is **blue**	Red	Red	Blue	Blue
6	3	Set the RAW-coloring field to **green**	Red	Green	Blue	Blue
7	4 (read from PE2)	Block until the RAW-coloring field is **green**	Green	Green	Blue	Blue
8	4	Set the WAR-coloring field to **purple**	Green	Green	Blue	Purple

Table 3: The dynamic check and set of the RAW-coloring and WAR-coloring fields during execution

forced nonetheless in actuality to be executed in the semantically correct order of (1, 2, 3, 4) with the help of static coloring information.

4.3 Further implementation considerations

One of the fundamental implementation considerations is the number of bits necessary to encode the dependence information in order to prevent possible semantic violations, that have been discussed in the previous two subsections. A further aspect that necessitates consideration is the number of communication registers to be shared between adjacent PEs. This architectural parameter is highly related to the frequency of global dependences of each application. While increasing the number of communication registers may be seen as desirable and providing insurance against application variance, nonetheless, one needs to consider that not only an area but also an indexing cost is thus incurred. For instance, if the size of a communication register file is 16, 6 bits are necessitated to encode each reference to the communication registers, with 4 bits for indexing and 2 bits for the reference coloring scheme. Consequently, it may be prudent to consider a reduced register file size, as long as performance is not degraded. An important role befalls compiler technology in order to provide the appropriate ordering to the communicating references, thus limiting the size of communication register file. For most applications, register file size can be seen to be constrainable to no more than 8, and frequently further down to 2.

In our framework, communication registers are only shared between adjacent PEs, as presented in Figure 1. On one hand, this results in extra **move** instructions that are necessitated to transfer the results produced by one processor to another, which may cause performance degradation. On the other hand, this simplifies interconnect routing complexity and the dynamic global dependence checking mechanism. In fact, if a communication register is shared by more than two PEs, **two bits** in total would not suffice for the *WAR-coloring* and *RAW-coloring* field of each communication register. While these two aspects constitute a design tradeoff, the compiler can schedule a pair of dependent instructions on the same PE or on adjacent PEs as much as possible during code generation, thus minimizing the number of extra **move** instructions. In the case where extra **move** instructions can not be avoided, the compiler can also schedule them early in order to hide the latency of bringing that value from the producer PE to the consumer PE.

5. PERFORMANCE EVALUATION

In this section, we provide a mathematical evaluation framework of the proposed register-level communication technique, since the commercial absence of the associated compiler technology precludes a precise analysis of the ensuing benefits. As was mentioned in Section 2.1, the framework on which our communication technique can be applied is quite flexible, with the number of internal PEs able to vary across architectural instantiations, with PEs being superscalar, or VLIW processors, or a mix of both. With no loss of generality, in this section, we use an example configuration consisting of four PEs, all of which are homogeneous 4-way processors. However, the choice of the processor type for internal PEs has a non-negligible impact on the performance of the register-level communication scheme we propose.

In the case where internal PEs are VLIW processors, independent operations are scheduled and packaged by the compiler into a long instruction that can be executed on a PE directly. This implementation is efficient in terms of hardware complexity and power consumption. However, an implicit synchronization is imposed for each long instruction: if one operation cannot finish, the complete VLIW instruction has to be stalled. These unnecessary stalls will further propagate to other PEs through global dependences, introducing unnecessary performance degradation. An example is presented in Figure 3, in which a load instruction on PE1 runs into a cache miss, causing the whole long instruction as well as all the following instructions on PE1 to be stalled. Stalls are further propagated to PE2, since one instruction on PE2 has a global dependence on that stalled long instruction on PE1. Similarly, PE3 and PE4 also need to be stalled for exactly the same reason.

On the other hand. in the case where internal PEs are superscalar processors, dynamic out-of-order execution is performed on each PE. Although the hardware complexity and power consumption in this case are less efficient, no performance degradation is caused by unnecessary propagated stalls. The only unnecessary performance degradation in this case is introduced by the disabling of register renaming on communication registers, which forces unnecessary stalls in the case of *WAR dependence*. If a read operation to a communication register is blocked, a subsequent write operation to the same communication register also needs to be blocked.

Since statically each PE is scheduled to perform its own execution flow smoothly, performance degradation is only caused by unpredictable run-time events, such as cache misses and branch mispredictions. For both VLIW and superscalar PEs, the potential performance degradation is a function of the frequency of global dependences and the penalty of run-

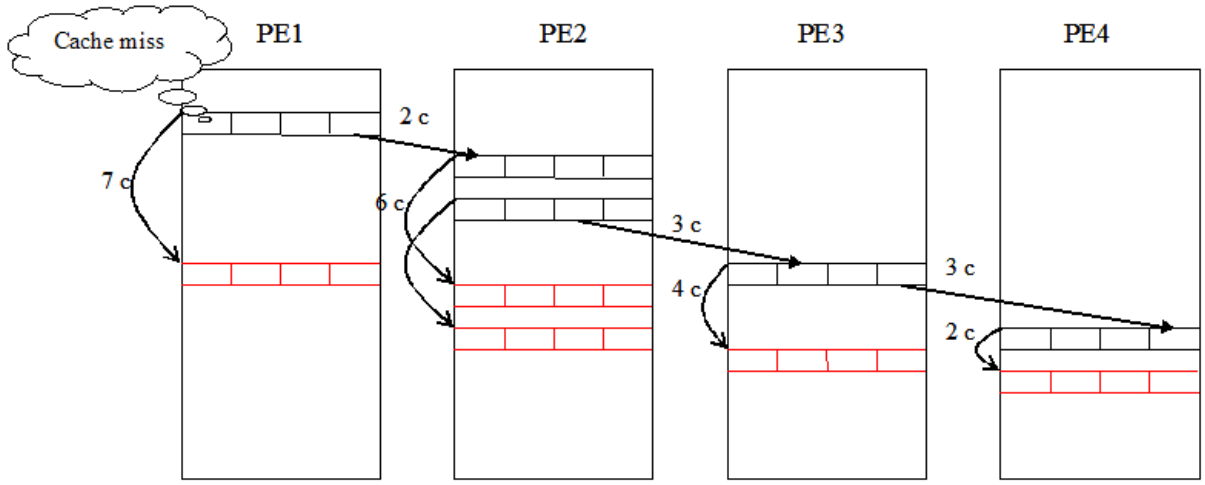

Figure 3: Propagation of stalls though global dependence chains

time events. Additionally, since the performance degradation of superscalar PEs is caused by the *WAR dependence* of communication registers, it is a function of the number of communication registers available between adjacent PEs. For VLIW PEs, the performance degradation is caused by the propagation of unnecessary stalls, which is also a function of the originally scheduled distance between the globally dependent instructions. To be more concrete, suppose in Figure 3 the cache miss on PE1 will cause the whole execution flow on PE1 to be stalled for 7 cycles. Assume that each long instruction needs one cycle to complete execution. Since the instruction on PE2 that has global dependence on PE1 originally is scheduled 2 cycles later, it can tolerate a stall of 1 cycle. As a result, the whole execution flow on PE2 needs to be stalled for 6 cycles. By the same reasoning, PE3 needs to be stalled for 4 cycles, and PE4 needs to be stalled for 2 cycles. In the following two subsections, two mathematical models have been developed to evaluate the potential performance degradation as a function of the aforementioned factors for VLIW PEs and superscalar PEs, respectively.

5.1 PEs are VLIW processors

For VLIW PEs, we have developed a mathematical model to reflect potential performance degradation as a function of the penalty of run-time events and the distance between the globally dependent instructions; the results can be observed in Figure 4. To simplify the model, it is assumed that each long instruction needs one cycle to finish execution. The values in Figure 4 represent the average number of cycles needed to be stalled per instruction. These values are normalized to the worst case results, i.e., the case when every instruction needs to be stalled. It can be observed from Figure 4 that the traditional VLIW processor constitutes the worst case (the second column), since that model implies an implicit synchronization for all four PEs, causing them to stall for the same number of cycles as the penalty of run-time events. On the other hand, the best case (the first column) is that each PE is executed independently and that there exists no global dependence. In this situation, only the original PE that generates a cache miss or branch misprediction needs to be stalled, while the other three PEs can progress normally. As a result, the performance degradation is 1/4. The other 3 columns represent the cases in which the average distances between two globally dependent instructions are 3, 6, and 11, respectively.

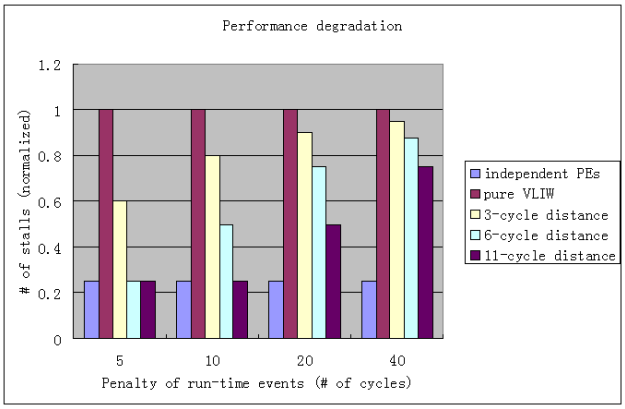

Figure 4: Average number of stalls per instruction (normalized) when PEs are VLIWs

It can be easily seen in Figure 4 that the proposed architecture is consistently superior to pure VLIW processors. The reason is that when a penalty of run-time events is propagated to other PEs through global dependences (if they exist), the number of cycles that need to be stalled is reduced because of the stall toleration capability enabled by the existence of slack between globally dependent instructions. In addition, it can be observed from Figure 4 that enlarging the distance between two globally dependent instructions enables the toleration of more cycles of stalls, thus in turn reducing the average performance degradation. However, enlarging the distance incurs a delay in the usage of the data, which causes performance degradation in the case when the unpredictable run-time event fails to occur. Therefore, the optimal distance between two globally dependent instructions depends on the occurrence possibility of unpredictable run-time events, which can be obtained straightforwardly through static profiling of the particular application.

5.2 PEs are superscalar processors

In the case where internal PEs are superscalar processors, the unnecessary performance degradation is a function of the frequency of global dependences, the penalty of run-time events, and the number of communication registers available between adjacent PEs. We have developed an additional mathematical model to reflect this relationship, as presented in Figure 5. To simplify the model, it is assumed that the reuse of communication registers is performed in a first-in-first-out (FIFO) order, and that each long instruction needs one cycle to finish execution.

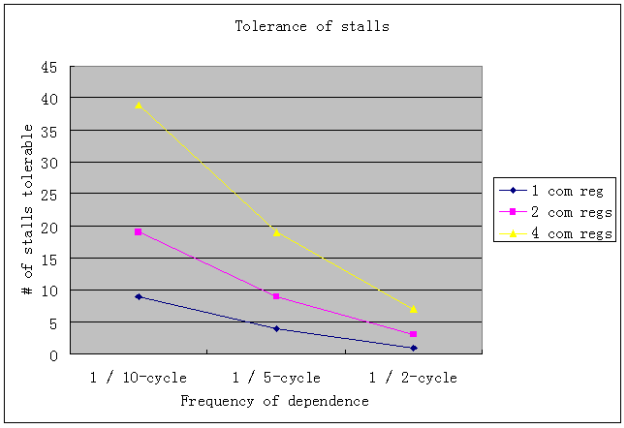

Figure 5: Average number of stalls that can be tolerated between adjacent superscalar PEs

The values in Figure 5 represent the average number of stalled cycles that can be tolerated between adjacent PEs. For a particular combination of communication register number and frequency of global dependences, if the number of tolerable stalled cycles is larger than the penalty introduced by an unpredictable run-time event, that unpredictable run-time event will introduce no unnecessary performance degradation in the average case. In Figure 5, it can be observed that the average number of stalled cycles tolerable is proportional to the number of communication registers. However, this ability to tolerate decreases significantly as the frequency of the global dependences grows. Although increasing the number of communication registers can reduce the potential performance degradation, from Figure 5 it can be seen that reducing the frequency of global dependences is more critical.

5.3 Overall evaluation

While the hardware architecture is herein described, the absence of the associated compiler technology precludes a precise analysis of the ensuing benefits. Yet even the results of the mathematical analysis, which constitute a worst-case bound of the expected benefits, as they rely on the assumption of a traditional compiler technology, evince substantial improvements. Future plans include the construction of an associated scheduling algorithm to be used for achieving efficient code generation during the compilation process. Such a compiler can be complemented by a simulator specific to the proposed microarchitecture and used for measuring the performance of the proposed framework.

6. CONCLUSION

In this paper, we have described a cheap yet efficient register-level communication framework to accelerate coupled embedded applications, achieved by allowing adjacent PEs to share a small register file. This simple organizational approach not only reduces the control and communication overhead associated with global storage architectures, but also improves the exploration of potential parallelism within complex embedded applications. The synergistic collaboration between the compiler, responsible for inserting communications as necessary and minimizing the influence on performance by reducing the number of global dependences, and the novel proposed architecture provides a framework for embedded system implementations on an SOC. A distributed dynamic checking mechanism, employed to handle unpredictable run-time events, assures prevention of potential semantic violations through the use of a novel reference coloring scheme within a negligibly small amount of hardware overhead. The proposed scheme enables extending drastically the realm of embedded applications through novel hybrid multi-processor architectural principles.

7. REFERENCES

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM 360 model 91: Processor philosophy and instruction handling. *IBM J. Research and Development*, 11(1):8–24, January 1967.

[2] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *Proc. of the 24th ISCA*, pages 206–218, June 1997.

[3] P. Faraboschi, J. A. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proc. of the IEEE*, 89(11):1638–1659, November 2001.

[4] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, September 1996.

[5] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. *Proc. of the 19th IPDPS*, pages 28a–28a, April 2005.

[6] S. Amarasinghe, et. al. Hot compilers for future hot chips. *presented at Hot Chips VII*, Stanford, CA, 1995.

[7] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *Proc. of the 22nd ISCA*, pages 414–425, June 1995.

[8] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A multilevel computing architecture for embedded multimedia applications. *IEEE Micro*, 24(3):56–66, May 2004.

APPENDIX

A. PROOF OF THE REFERENCE COLORING ALGORITHM

If we order all the accesses to a particular communication register, it is easy to find that the proposed coloring algorithm will iterate to use exactly the same RAW and WAR color every four instructions. This implies that the i^{th} instruction will always have the same RAW and WAR color as the $(4n+i)^{th}$ instruction for integer n.

Based on this observation, the following lemma can be proved:

LEMMA 1. *According to the proposed static scheduling and coloring algorithm, it is impossible to have two adjacent accesses to a particular communication register emanating from the same PE that have the same RAW and WAR color.*

PROOF. According to the previous analysis, the two closest instructions that have the same RAW and WAR color are instruction i and instruction $i+4$. If instructions i and $i+4$ are two adjacent accesses on the same PE to that particular communications register, all the 3 instructions between them that display different RAW and/or WAR colors, that is, instructions $i+1, i+2, i+3$, are scheduled onto another PE. The operations performed by instructions $i+1, i+2, i+3$ consist of either *read, write, read* operations, or *write, read, write* operations. In both cases, a *write* operation to the communication register is followed by a *read* operation emanating from the same PE, constituting a redundant usage. Since all the redundant usages should have been eliminated during code generation, evidently, this case is impossible, implying that it is impossible to have two adjacent accesses to a particular communication register emanating from the same PE that have the same RAW and WAR color. □

Based on Lemma 1, the following theorem can be proved:

THEOREM 1. *With the help of the dynamic check and set of the **RAW-coloring** field and the **WAR-coloring** field, the reference coloring algorithm will guarantee semantic correctness during dynamic execution as long as each PE will not reorder the instructions that access the same communication register on its own.*

PROOF. The dynamic checking and setting mechanism provides a constrained environment for the execution of an instruction with global dependences. According to the previous analysis, the same environment repeats every four instructions. As a result, in the case of semantic violation, instruction i is either scheduled prior to instruction $i-3$, or subsequent to instruction $i+3$. According to Lemma 1, at least one of instructions $i+1, i+2, i+3$ is scheduled on the same PE as instruction i. As long as each PE is not allowed to reorder the instructions with global dependences on its own, instruction i cannot be dynamically scheduled subsequent to instruction $i+3$. In an analogous manner, instruction i cannot be dynamically scheduled prior to instruction $i-3$, either. □