CDE: A Compiler-driven, Dependence-Centric, Eager-executing Architecture for the Billion Transistors Era

Carmelo Acosta [1], Sriram Vajapeyam [2], Alex Ramirez [1], Mateo Valero [1]

[1] Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
{cacosta,aramirez,mateo}@ac.upc.es

[2] Independent Consultant
Surathkal, India
sriram@cs.wisc.edu

Abstract – *We propose an evolutionary new approach to high-performance processor architectures that can scale to use increasing numbers of transistors while meeting the constraints of design complexity, wire lengths, and power management. The fundamental idea is to make the architecture and the micro-architecture dependence-centric, in contrast to prior instruction-centric and trace-centric processors. We rely on the compiler for much of the program partitioning and parallel execution management, thus creating a compiler driven dependence architecture (CDE). Complexity-effective high performance is achieved via hierarchical program partitioning and matching hierarchical hardware. The compiler partitions program binaries into "horizontal" control epochs (structures such as super-blocks, nested loops, etc.) that in turn contain several "vertical" dependence clusters (DCs). The CDE hardware consists of a cluster of simple single-issue cores that execute individual DC threads, and a special epoch processor core (EPC) that fires up these DC threads by processing the higher-level epochs. The EPC uses hardware support to speculatively sequence through and quickly process the epochs, whereas within each epoch, the compiler handles all ILP meta-operations such as rename, squash, redo, commit, etc. in software, in a distributed fashion on the processing element cluster. The compiler enhances ILP and communication locality by employing VLIW-style trace scheduling and selective eager execution within an epoch. Several benefits can accrue from the combination of features in CDE, with regard to complexity effectiveness, lower ILP overheads, efficient exposure of far-flung ILP, reduced branch mis-prediction penalty, scalability to match resources, etc.*

Keywords **–** Dependence Architectures, Program Partitioning, ILP Compilation, VLIW, Clustered micro-architectures, Scalability, Complexity-Effectiveness, Hardware-Software Co-design

1. Introduction

High-performance processor architecture's challenges in the emerging billion-transistor era lie in effectively utilizing the free transistors under the constraints of reasonable design complexity, short wire lengths, and power management. The architectural goals could be high performance for a particular workload class or good performance across multiple workload classes. Different design philosophies exist for achieving these goals and meeting the constraints. While an entirely software based approach for extracting performance would eliminate hardware complexity and work very well for particular workload classes, other workloads require the exploitation of runtime information (either by the hardware or a dynamic optimizer) to achieve high performance. Clustered, possibly hierarchical hardware is being accepted as a way of dealing with the wire-length/wire-delay and design complexity issues.

The challenge of designing clustered, hierarchical hardware and corresponding execution models has spawned several different approaches. RAW microprocessors [Waingold97] rely almost entirely on the compiler to orchestrate intra-basic-block ILP and communication on a cluster of simple MIPS R2000-like processors

interconnected by compiler-exposed communication channels. The Grid Processor Architecture (GPA) [Burger01] uses a combination of software and hardware to orchestrate both intra-hyper-block ILP and speculative cross-hyper-block ILP on a cluster of ALUs interconnected by a router-based multi-hop grid. Multiscalar processors [Franklin93, Sohi95] use a combination of hardware and software to orchestrate ILP and speculative task-level parallelism on an ordered ring of processors interconnected by queues. Dependence-based architectures [Pala97] rely on hardware to steer chains of dependent instructions to individual PEs of a PE cluster. ILDP architectures [Smith02] extend this approach by relying the compiler to produce accumulator-based instructions that help hardware dependence steering. Trace Processors [Vajapeyam97, Rotenberg97] rely on hardware to form and dispatch traces to the individual PEs of a PE cluster.

We propose a novel approach to achieving high performance that combines a hierarchical partitioning of the program with a corresponding hierarchical design of the hardware. A central idea is to formalize the notion of a set of dependent instructions, and to provide it a central role in the architecture and micro-architecture. Further, we balance the roles of the compiler and the hardware in a new way, to achieve complexity effectiveness and high performance.

The CDE compiler hierarchically partitions program binaries into "horizontal" *control epochs* (e.g. super-blocks, nested loops, or larger structures) that in turn contain several "vertical" *dependence clusters* (DCs), each having multiple instructions. The compiler enhances ILP and communication locality within epochs by employing *VLIW-style trace scheduling* [Fisher81] and *selective eager execution* -- an epoch serves as a boundary for such compiler ILP optimizations. The CDE hardware has a scalable cluster of simple, MIPS R2000-like single-issue processor cores that execute individual DC threads. A special epoch processing core (EPC) processes summary information about an epoch to fire up the DC threads of the epoch (resident on individual cores). CDE exploits parallelism in a hierarchical fashion. The EPC speculatively sequences through epochs, thus overlapping the execution of different epochs (epoch-level parallelism, ELP). Within each epoch, individual DC threads execute in parallel.

ILP handling is hierarchically organized in CDE to achieve complexity effectiveness. At the epoch level, the EPC uses hardware support for register renaming, epoch speculation, etc. Internal to an epoch, the compiler handles *in software* all DC-level ILP meta-operations (such as squash, redo, commit, etc) via special inter-DC messages and processor core predicate bits. Importantly, these operations are applied to DC threads rather than to individual instructions, thus amortizing their overheads over the multiple instructions that belong to a DC thread.

Several benefits can accrue from the combination of features in CDE, with regard to complexity effectiveness, lower ILP overheads, efficient exposure of far-flung ILP, reduced branch misprediction penalty, scalability to match resources, etc. Further, while we have several ideas for making the hardware configurable and adaptable so as to be able to provide very good performance across a range of workloads, that aspect is beyond the scope of this paper.

The rest of this paper is organized as follows. In the next section, we describe the proposed architecture in more detail. We contrast our approach with previous work in Section 3. We discuss the current status of the work, important open design issues, and possible future lines of research in Section 4. We summarize and conclude in Section 5.

2. The CDE Architecture and Micro-Architecture

CDE uses a new approach to exploiting free transistors that hierarchically partitions programs and the hardware in co-designed fashion, relying heavily on the *compiler* for partitioning and scheduling, and on both compile-time and runtime *speculation* for keeping the transistors occupied. We believe that the following are important goals to achieve scalability of ILP and complexity effectiveness:

1. ILP meta-operations such as squash, redo, and commit should be made simple and relatively inexpensive in order to enable extensive speculation that consumes the available transistors to improve performance.

2. Both run-time and compile-time ILP exploitation mechanisms are required for high performance, and judicious partitioning of ILP handling between hardware and software is required to ensure complexity effectiveness.

To achieve the above goals, we use a *Dependence Cluster (DC)* as the fundamental unit around which we organize the architecture and the micro-architecture. Dependence clusters have several advantages: they minimize global communication, localize architectural state updates making squash, re-do, and commit easier, compress large virtual instruction windows into small physical windows, and simplify instruction issue. CDE uses traditional *VLIW-style trace scheduling* techniques, within manageable code boundaries, to obtains DCs that have better properties. Towards meeting both complexity-effectiveness and performance goals by exploiting free transistors, CDE uses compiler-orchestrated *selective eager execution* across difficult branches.

We now describe the CDE program partitioning scheme and the CDE architecture and microarchitecture design at some level of detail. First, we describe the hierarchical partitioning of a program by the compiler, and its use of trace scheduling and selective eager execution. Next, we describe the CDE architecture and micro-architecture.

2.1. Hierarchical Program Partitioning

Control Epochs. The CDE compiler first partitions the program into horizontal *control epochs* (such as superblocks, nested loops, or bigger chunks). An epoch serves multiple goals:

(i) it serves as the unit by which the epoch processing core (EPC) quickly advances through the program to speculatively expose far-flung ILP. Hence, the larger an epoch, the better; further, epoch-terminating branches should be highly predictable at runtime.

(ii) it determines the scope of the compiler's CDE optimizations and of eager execution. Thus an epoch should strike a balance between providing too little and too much scope for the compiler.

(iii) within an epoch, the hardware expects the compiler to handle all ILP meta-operations (e.g. squash, redo, commit, rename, etc.) in software, thus reducing hardware complexity.

Thus an epoch could be different from previously proposed code segments such as hyperblocks, superblocks, etc. Figure 1(a) shows a sample epoch from a SPEC benchmark program (and its further hierarchical partitioning). In this example, a doubly-nested loop is chosen as an epoch, as a reasonable satisfaction of the above goals. An alternative choice for an epoch would have been just the inner loop, in which case the hardware would have to step through individual iterations of the outer loop, resulting in smaller steps through the program.

VLIW-style traces. Within an epoch, the compiler identifies *VLIW-style traces* (Figure 1d) for further partitioning, to improve the properties of the resulting dependence-cluster partitions and to selectively eager execute alternate traces that encompass difficult branches. As shown in Figure 2-a, we embed hard-to-predict branches within an epoch and overcome their performance impact by using *selective eager execution, via* constructing alternative traces around these branches. The machine speculatively initiates execution of *both* traces (i.e. their corresponding dependence clusters), and one of the traces is squashed upon branch resolution (Figure 2-b). Our approach differs from a pure hardware approach that speculates down both paths [SEE98, DEE02], in that we can do global instruction scheduling and optimization of post-branch instructions with pre-branch instructions on *both* the paths. Our approach differs from the traditional VLIW compiler approach in that we construct alternative traces for both paths, instead of speculating on one trace and generating fix-up code for the alternate path.

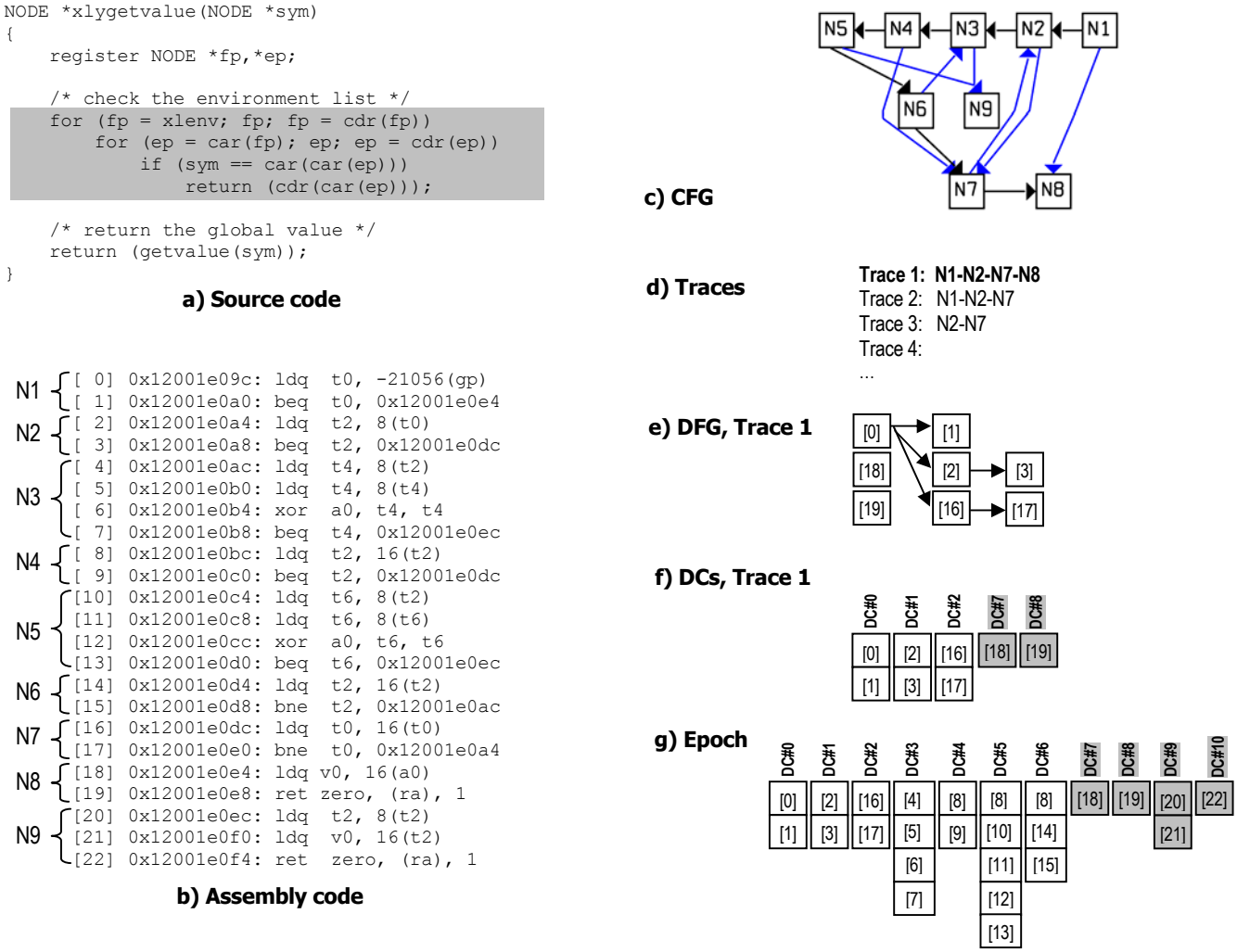

Figure 1. Hierarchical Program Decomposition into Control Epochs and Dependence Clusters

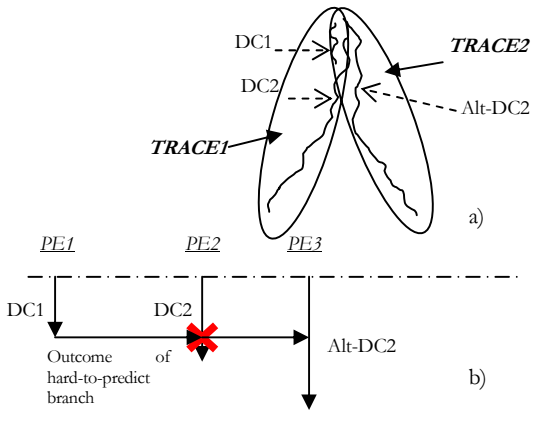

Figure 2.- Executing both paths beyond a branch by building alternative VLIW-style traces and initiating both DC2 and Alt-DC2.

Dependence Clusters (DCs). Each VLIW-style trace is partitioned into vertical *dependence clusters* (DCs, Figure 1f). Each DC is essentially a set of dependent instructions, but these instructions could also have some dependencies on other DCs since programs rarely offer very clean partitions of the data dependence graph. Inter-DC dependences is handled via explicit send/receive messages between DCs. The DC partitioning goals and heuristics attempt to strike the best balance between localizing most communication within a DC and exploiting ILP by placing independent instructions in separate DCs. We expect to find DCs having suitable properties by our use of VLIW-style traces and large control epochs, as opposed to just basic blocks or hyper-blocks.

DC-to-Processor-Core Mapping. Given the partitioning of an epoch into DCs, the compiler maps individual DCs to specific processing cores. The compiler assumes each core has an instruction memory; the hardware might implement this memory either as local memory or perhaps as a cache. Such static mapping of DCs to PEs considerably simplifies both the inter-DC communication that the compiler has to handle (as described next) and the epoch processing core (EPC).

Inter-DC Communication. Subsequent to DC partitioning, the compiler handles inter-DC communication. Explicit sends and blocking receives are inserted to implement inter-DC communication over a dynamic operand network. Note that the overall inter-instruction communication model available to the compiler is a hierarchical one: within each PE, a local register file (or set of accumulators) is visible to the compiler for the use of DC-local values. Inter-DC communication within an epoch is via direct sends and receives, and/or writes/reads to remote registers (local registers of other DCs). A global architectural register file is used for communication across epochs.

ILP meta-operations. The compiler also insets ILP meta-operations into the partitioned DCs, since the hardware does not provide ILP support within an epoch. ILP meta-operations such as squash, redo, commit, etc. are handled by the explicit sending (and forwarding) of messages from one DC to another within an epoch, or by the epoch processing core (EPC) to all the DCs of an epoch. For example, a DC that resolves a branch will send a squash message to the DCs of the wrong path corresponding to that branch outcome, resulting in those DCs self-destructing. The receipt of a redo message will reset the PC of the receiving DC. Where appropriate, the received message is further forwarded to other DCs have data dependencies on the current DC.

Code Generation. The program binary is organized hierarchically into epochs and DCs. As Figure 3 shows, each epoch is tagged with information about the live-in and live-out registers of that epoch, and contains instructions to fire up the threads of the epoch (as well as to commit them or squash them). Epochs are laid out like a traditional program binary in the address space, so that the EPC can sequence through the epochs in the same way that a regular processor sequences through a program. However, the DCs of the program can be laid out anywhere in the program address space since each of them is an independent thread (having some communication with other threads).

2.2. CDE's Hierarchical Hardware

CDE hardware is organized hierarchically corresponding to CDE's hierarchical program decomposition. This enables us to reduce complexity, deal with wire delays, and achieve scalability. A grid of simple MIPS R2000 like processing cores is orchestrated by a special epoch processing core (EPC, Figure 4). The EPC quickly and speculatively steps through epoch summaries (Figure 3), firing up the constituent DC threads on their pre-assigned processing cores. Each processing core independently executes the DC threads fired up on it by the EPC, and communicates with other PEs (DCs) as necessary via send/receive instructions over the processor grid, which is a dynamic operand network.

Epoch Processing Core (EPC). The top level of the hierarchy is an epoch processing core (Figure 4) that fetches and processes epoch summaries speculatively in program order, one at a time. The EPC renames the live-in and live-out registers of an epoch (listed at the head of the epoch, as in Figure 3), and sends start messages to the DCs of the epoch (which reside on compiler-assigned processing cores). CDE's architectural register file (ARF) is renamed by EPC onto a scalable, distributed physical register file (PRF) than has a few registers on each of the processing cores. Thus the PRF size scales with the number of processing cores in the CDE hardware. Observe

```
Epoch1

# Preamble
Live-Ins:  R1, R4, R9, R12
Live-Outs: R3, R7, R9, R14

# Thread Fire-up Code
Send  PE4, <Epoch1.DC1, rename-map, START>
Send  PE7, <Epoch1.DC2, rename-map, START>
Send PE11, <Epoch1.DC3, rename-map, START>
Send PE12, <Epoch1.DC4, rename-map, START>

# Thread Commit/Squash Code
# Predicate = Commit/Squash
Send  PE4, <Epoch1.DC1, predicate>
Send  PE7, <Epoch1.DC2, predicate>
Send PE11, <Epoch1.DC3, predicate>
Send PE12, <Epoch1.DC4, predicate>

# Epoch Branch Code
BranchConditional CondReg1, Target-Epoch
```

Figure 3. Code for a control epoch. EPC Hardware renames the live-ins and live-outs specified at the epoch's beginning, and subsequently the EPC executes the thread fire-up instructions (send instructions) specified in the epoch. When necessitated by an epoch-level mis-prediction, the EPC has to execute the thread squash instructions. EPC design is as yet an open issue.

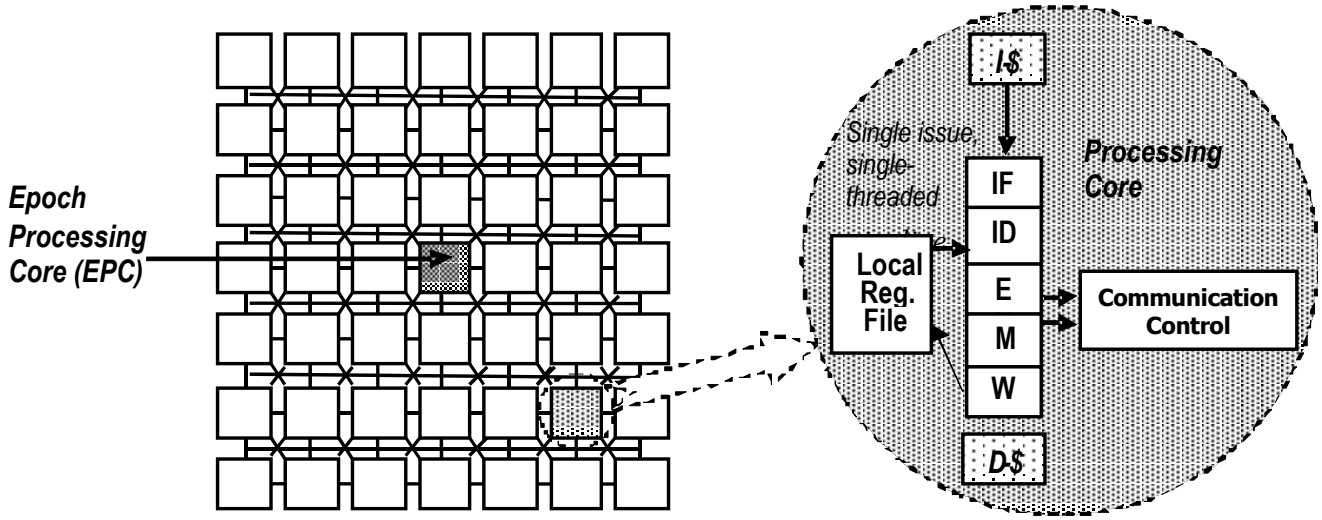

Figure 4. CDE's Hierarchical Hardware. A grid of self-sufficient, MIPS R2000-like cores is orchestrated by a special epoch processing core (EPC). The processing cores have a small communication unit to handle both dynamic operand routing and the lightweight processing of incoming messages. The EPC's detailed design is as yet an open issue.

that each processing core has two sets of registers: one set being part of this global PRF, and the other being the local register file (or accumulators) used by a DC's local values.

Subsequently the EPC speculatively branches to the next epoch in program order and continues with such processing, before the first epoch completes execution, similar to speculative branching in a superscalar processor. (Notice the presence of some branch code in each epoch summary to handle epoch level control flow.) When speculation goes wrong, the EPC needs to execute the thread squash code (Figure 3) present in each epoch that is to be squashed.

Individual Processing Elements (PEs). Each processing element in the cluster is a MIPS R2000-like simple, self-sufficient, in-order single-issue processor. Each PE has local instruction and data memories, local control logic, and a tiny communication processor. The latter consists of a small router to handle operand routing on the dynamic operand network and a tiny message processor to receive incoming messages and process them without interrupting the running thread. The message processing hardware simply stores each incoming message in the appropriate place after decoding the message.

A PE fetches its DC thread's instructions one at a time from local RAM, essentially assuming that each instruction is dependent on the previous one. The PE uses a simple register reservation scheme to handle instructions of unpredictable latency. Each core has a local register file as well as a few registers of the global register file (to which the ARF is mapped). When a thread starts execution, the thread's rename map (received from the EPC along with the start message) is loaded on to the PE's map table. All references to the architectural register file look up this local map table and use the corresponding physical register identifier. The PE hardware is enhanced to implement send and blocking-receive instructions. Overall, each core is fairly simple, and executes one chain of dependent instructions (DCs) at a time.

Design Considerations. The overall goal of hardware design is complexity effectiveness, scalability, and sensitivity to wire delay. However, some design decisions need justification. We use special hardware in the EPC so as to speed up epoch level processing and thus expose far flung ILP more quickly (and speculatively). The thread fire up mechanism involves sending messages from a central point to all the PEs, and thus perhaps multiple hops (cycles) through the grid. We expect to cover this latency by the epoch-level parallelism exposed by the EPC. Further, each thread start message requires the register rename table (or the appropriate entries of it) to be sent to the corresponding PE. We expect to use techniques such as multicast to reduce this overhead, and also expect the exposed parallelism to cover the latency.

3. Comparison with Previous Work

Several different approaches have been proposed to address the issue of modular, hierarchical hardware and corresponding execution models. CDE proposes a new way of addressing these challenges, as briefly described in the introduction. We now discuss previous work that is most relevant to CDE. Table 1 highlights the key differences between CDE and other approaches.

RAW microprocessors [Waingold97] attempt to achieve high performance and complexity effectiveness by handling most of the ILP operations in software. RAW relies on the compiler to parallelize code and schedule it onto a grid of simple MIPS 2000-like processors, and compiler-exposed communication channels. Each basic block is parallelized onto the PE cluster [Lee98]; individual PEs independently fetch and execute their instructions, and communicate directly with other PEs over the scalar operand network in pre-programmed fashion. Most of RAW's reported performance benefits seem to come from loop parallelism [Lee98].

The UT-Austin Grid Processor [Burger01] appears to borrow heavily from the RAW philosophy, but applies certain concepts at the ALU level instead of the processor level to deal with wire delays at a finer granularity. The GPA grid consists of simple ALUs instead of processors, and the compiler schedules instructions from a hyper-block onto the ALUs of the grid, with result forwarding between instructions (ALUs) being explicitly programmed onto the grid communication channels. The instructions of a hyperblock are fetched in one shot

from a shared I-cache, and instructions assigned to a computation node are issued in data-flow fashion by the local hardware. Multiple hyperblocks can be speculative multiplexed onto the GPA grid; in addition, multiple grid processors can be built on a chip [Sankar03] to achieve TLP. Thus the GPA uses some combination of software and hardware in its attempt to build a high-performance architecture.

Multiscalar processors [Franklin93, Sohi95] use a combination of hardware and software. The compiler partitions the program into horizontal tasks (e.g. a loop or a procedure); hardware speculates across the tasks, with each processor in a ring executing one task and communicating data with others over a ring network. A task is assumed to capture a good amount of dependences, and thus is used to both localize dependences and walk through the program in larger steps.

The ILDP approach [Smith02] extends the more traditional hardware-only, dependence-based approach [Pala97] to enhance complexity effectiveness. Instructions fetched and decoded by a traditional pipeline are steered to individual PEs in a cluster based on their dependences [Pala97]. An accumulator-based ISA is used in ILDP to make such steering simpler and to allow the compiler to expose dependencies. An individual PE thus executes a sequence of dependent instructions that only occasionally communicate with other PEs.

Trace Processors [Vajapeyam97, Rotenberg97] employ a level of execution hierarchy to achieve complexity effectiveness and high ILP. They fetch and dispatch traces (of say 16 instructions) to a PE cluster, thus achieving higher instruction dispatch bandwidth than other hardware approaches. Since traces are approximations of dependence clusters, hierarchical communication clustering is achieved along with a distributed large instruction window.

	CDE	RAW	GPA	Multiscalar	Trace Processor	ILDP
Hierarchical ILP Exploitation						
Program Partitioning	Epochs, DCs, Instructions	Basic Blocks, Instructions	Hyper-blocks, Instructions	Tasks, Instructions	Traces, Instructions	Instructions (accumulator-based)
Hierarchical HW	Yes – EPC and PEs	No – Just PEs	Yes – ALUs and Hyper-block Sequencer	Yes – Task Dispatcher and PEs.	Yes – Trace Dispatcher and PEs	Yes – Instruction Steerer and PEs
PE Complexity	MIPS R2000-like	MIPS R2000-like	Single ALU + a few reservation stations	MIPS R2000-like	Functional units set, trace buffer, & register file	Functional units set, dependence queue, & accumulators
Epoch size (largest code unit used for program sequencing)	Some large segment (e.g. nested loops)	Instructions	Hyper-block	Task: typically, a loop or a procedure	Traces – runtime instruction sequences	Instructions
Scope of Compiler Opts	Epoch	Basic block (hyper/super blocks?)	Hyper-block	Procedures or larger code segments	N/A	Procedures?
Program Memory						
Code-mapping to PEs	Static	Static	Static	Static/Dynamic	Dynamic	Dynamic
RAM	Distributed across PEs	Distributed across PEs	Centralized	Distributed across PEs	Centralized	Centralized
Instruction	Distributed,	Distributed,	Centralized	Distributed,	Centralized	Centralized

Fetch	one at a time per PE	one at a time per PE		one at a time per PE		
Grouping of dependent instructions	Yes, static – compiler forms DCs	Yes, static – compiler maps to PEs	No – compiler spreads dep. instructions across ALUs	Task is used as an approximation	Trace is used as an approximation	Yes, static – compiler maps them to accumulators

Communication

Communication localization (to within a PE)	Via DCs	Via code mapping	Instructions distributed across ALUs to match communication pattern	Via tasks	Via Traces	Via Dependency Chains
Inter-PE Communication scheduling (operand network)	Dynamic, via direct sends and receives over grid	Static, via direct sends and receives over grid	Dynamic, via writes to remote registers over grid	Dynamic, via remote writes over inter-PE queues	Dynamic, via global register file	Dynamic, via global register file
Communication across Epochs	Distributed Register File	Distributed Register File	Centralized Register File	(Same as above)	(Same as above)	(Same as above)

Control Flow

Epoch-level Control Flow	EPC speculates	Non-speculative, all PEs synchronize	Hyper-block speculator	Task speculator	Trace speculator	Instruction speculator (branch predictor)
Intra-Epoch Control Flow	Selective eager execution across DCs	Local branching within each PE	No branches – "execute-all" approach	Super-scalar model	No branch – trace captures dynamic path	No hierarchical control flow

Table 1. A comparison of CDE with other approaches to modular, hierarchical hardware and a program corresponding execution model.

4. Discussion

The CDE architecture is in the early phase of its definition, and thus detailed performance evaluation is premature. We have manually studied important portions of some SPEC benchmarks to help design aspects of CDE. We first present a small illustration of how DCs would execute on CDE's PE cluster, and then discuss some significant aspects of CDE execution.

Figure 5 presents brief snapshots of program execution on CDE. Epoch execution starts in the EPC by the fetching of the epoch summary (Figure 3). The EPC renames the live-in and live-out registers of the epoch, and then executes the thread start instructions (the *send* instructions in the thread fire-up section of the epoch summary) to fire-up individual DC threads on their pre-assigned PEs in a portion of the PE cluster (Figure 5b). The compiler will assign an epoch's DCs to neighboring PEs so as to match the inter-DC communication pattern and thus reduce communication latency. The EPC *speculatively* branches to the next epoch summary in the program's control flow and fires up its DC threads without waiting for the previous epoch to complete, thus exposing epoch-level parallelism. Figure 5d shows a second epoch assigned to a neighboring portion of the PE cluster. Note that the compiler can assign epochs to nearby sub-clusters based on expected dynamic program

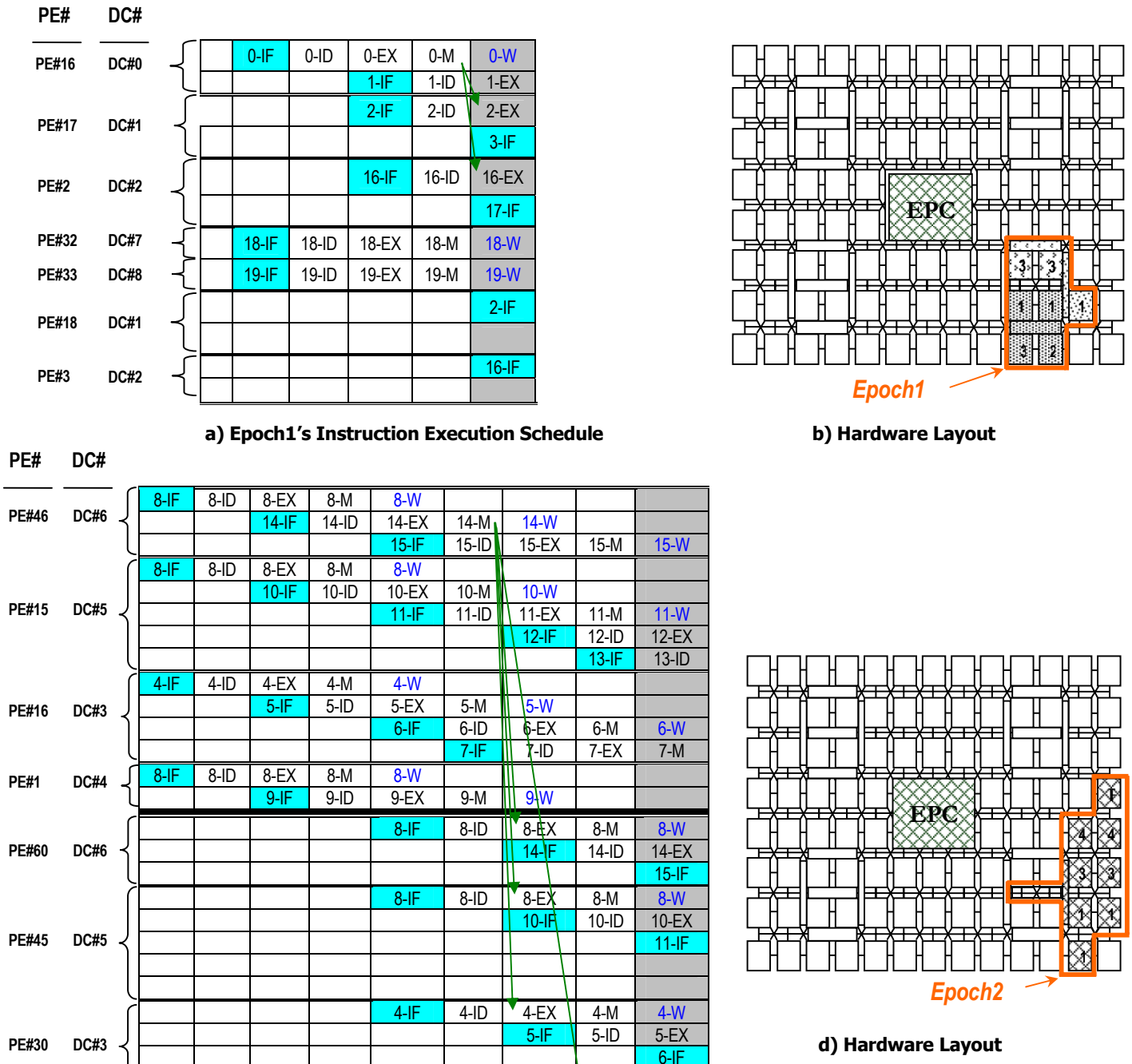

Figure 5. Example code execution on the CDE architecture.

control flow, so as to minimize inter-epoch communication delays (inter-epoch communication occurs via the distributed global register file).

An epoch's DCs start execution within their respective PEs after a communication delay from the EPC to the PEs. Consider the sample execution schedule from Epoch1 shown in Figure 5a:

- *Cycle 1*: Initial EPC– PE communication delay. This would be multiple cycles in practice, because of the multi-hop grid network.

- *Cycle 2*: DC#0, DC#7 and DC#8 start execution on their respective PEs. Note that neither DC#1 nor DC#2 start execution since they have data dependencies (see arrows in Fig. 5-a) with DC#0.

- *Cycle 3:* Execution continues. Notice that DC#0's second instruction is not fetched on this cycle, because the compiler indicates that it is dependent on the first instruction, a memory operation of multi-cycle latency.

- *Cycle 4:* This is the cycle where DC#1 and DC#2 start execution, i.e. their first instructions are fetched for processing, and happen to match data availability on clock 6. If a global clock is present, then a combination of compiler and hardware support can achieve such "just-in-time" scheduling of DCs within an epoch in CDE. Also note that it is in this cycle that the second instruction of DC#0 is fetched, since it is dependent on the previous memory instruction of that DC. Thus CDE can achieve "just-in-time" fetch with a combination of compiler and hardware support.

- *Cycle 5:* Register t0 is generated in PE#16 and sent to PEs #17 and PE#2. Besides, t0 is bypassed within the same PE (in the same functional unit) to the DC's next instruction (Instruction 1).

- *Cycle 6:* Register t0 arrives at PEs #17 and PE#2 and the executions of the DCs continue

Figure 5c illustrates a similar schedule for DCs from a second epoch, and Figure 5d shows a neighboring portion of the PE cluster having been assigned to this epoch.

We now observe several features of CDE execution. ILP is exploited hierarchically. Different epochs can be executing speculatively in different parts of the PE cluster. The different DCs within an epoch execute in parallel, and can *slip* with respect to each other in terms of instruction fetch and execution. Register values are speculatively communicated across epochs via the global physical register file, thus enabling the slip to span epochs. We expect that this will result in CDE exploiting a similar kind of parallelism as speculative multi-threading. The use of good size epochs will enable the EPC to expose far-flung parallelism, especially since epochs are de-limited keeping control flow aspects also in mind. With regard to ILP overheads, we observe that individual squash, commit, and redo operations are carried out on entire DCs rather than individual instructions, and this amortizes the cost of the operations over all the instructions of a DC. Our use of a DC-centric architecture is fundamental to this performance benefit. (This could also result in power savings compared to alternative ILP approaches.) Further, CDE uses speculative eager execution to handle difficult branches. This is a good tradeoff of free transistors for performance. Also, this approach is scalable in that the compiler can increase the extent of selective eager execution to match the available transistors.

Open Issues and Caveats. As mentioned earlier, CDE design is in its preliminary stages. We have not directly addressed the memory architecture issues for CDE, in particular how to use the RAM we assume present in each PE. The memory disambiguation scheme will have a significant impact on CDE performance. More importantly, the choice of suitable epochs, the compiler's ability to find such epochs, and the characteristics of DCs within these epochs will all have fundamental bearing on CDE's performance. However, previous work indicates several factors in favor of CDE: (i) there is considerable far-flung ILP in general-purpose programs; (ii) VLIW-style trace scheduling yields significant benefits, and we hope, DCs have good characteristics; (iii) within the relatively limited scope of an epoch, as opposed to whole-program analysis, the compiler could do a good job of exploiting ILP.

5. Summary and Conclusions

We have proposed a dependence-centric architecture, CDE, as a new way of achieving high performance in a complexity effective way. By formalizing the notion of a dependence cluster (DC) and providing it a central role in both the architecture and the micro-architecture, we enhance parallelism while simultaneously reducing the overheads of managing the parallelism. A DC becomes the fundamental unit of ILP meta-operations such as squash, redo, and commit, thus amortizing the overheads of those operations over the multiple instructions of the DC. This increased efficiency in handling mis-predictions in turn enables us to more easily deploy speculation, especially selective eager execution. In turn, this provides us a scalable way of exploiting free transistors to improve performance.

CDE relies on the compiler to partition programs hierarchically into horizontal control epochs that contain the vertical dependence clusters. Within each control epoch, the compiler handles all ILP meta-operations in software, thus achieving complexity effectiveness. CDE hardware is hierarchically organized into an Epoch Processing Core (EPC) and a cluster of MIPS R2000-like processing elements (PEs). The EPC speculatively steps through the epochs, firing up the DC threads of each epoch in the PE cluster. This results in hierarchical ILP exploitation: speculative epoch-level parallelism and compiler orchestrated DC-level parallelism.

Our approach might be viewed as a suitable adaptation of the philosophies of both VLIW machines and superscalar processors to tomorrow's issues of free transistors, design complexity, wire lengths, and power budgets. Instead of very long instruction words, we build very long dependence clusters from VLIW-style traces. Dependence clusters help remove traditional VLIW and superscalar limitations by de-coupling instruction fetch and issue of the different issue slots of a clock cycle. This should also enable CDE to exploit the kind of parallelism exploited by speculative multi-threading. Free transistors allow us the luxury of (selectively) building both the alternative traces that span a difficult branch (subject to the limitations of the memory system), providing an equivalent of complex runtime speculation. At the same time, we use runtime speculation support at the higher level of epochs, to help expose far-flung ILP. In contrast to previous approaches, we strike a new balance between the compiler and hardware, with the compiler responsible for intra-epoch ILP and the hardware for inter-epoch ILP. The CDE architecture shares different interesting features with different previous proposals such as RAW, GPA, ILDP, Trace Processors, etc., while having many novel features of its own.

CDE's design is currently in the preliminary stages. An important caveat is that CDE's performance is critically dependent on the characteristics of control epochs and dependence clusters that the compiler is able to identify in real programs. We are currently studying program characteristics, and intend to develop both the compiler and a hardware simulator for detailed performance evaluation of CDE. We expect the novel and unique combination of features in CDE to provide several benefits over previous approaches to high performance, complexity effective architectures.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIC-2001-0995-C02-01, CEPBA and an Intel Fellowship. Carmelo Acosta is also supported by the Ministry of Science and Technology of Spain grant BES-2002-0015.

References

[Burger01] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, Stephen W.Keckler. *A Design Space Evaluation of Grid Processor Architectures*. MICRO 2001. pag 40-51.

[DEE02] Augustus K. Uht. *Disjoint Eager Execution: what it is / what it is not.* ACM SIGARCH CAN. pag 12-14 .March 2002.

[Fisher81] J.A. Fisher. *Trace scheduling: A technique for global microcode compaction*. Transactions on Computers, C-30(7). pag. 478–490. July 1981.

[Franklin93] M. Franklin. *The Multiscalar Architecture.* Ph. D. Thesis, Computer Sciences Technical Report #1196, University of Wisconsin-Madison, Madison, WI 53706, November 1993.

[Lee98] Walter Lee et al. *Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine*. ASPLOS 1998. pag 46-57.

[Pala97] Subbarao Palacharla, Norman P. Jouppi, James E. Smith. *Complexity-Effective Superscalar Processors*. ISCA 1997. pag. 206-218

[Rotenberg97] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, Jim Smith. *Trace Processors*. MICRO 1997. pag 138-148.

[Sankar03] K. Sankaralingam, *et al.* Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. ISCA 2003.

[SEE98] Artur Klauser, Abhijit Paithankar, Dirk Grunwald. *Selective eager execution on the PolyPath architecture*. ISCA'98. pag 250-259.

[Smith02] Ho-Seop Kim, Smith, J.E. *An instruction set and microarchitecture for instruction level distributed processing*. ISCA 2002. pag 71 – 81.

[Sohi95] Gurindar S. Sohi, Scott E. Breach, T. N. Vijaykumar. *Multiscalar Processors.* ISCA 95. pag 414-425.

[Vajapeyam97] Sriram Vajapeyam, Tulika Mitra. *Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences*. ISCA 1997. pag. 1-12

[Waingold97] E. Waingold et al. *Baring It All to Software: Raw Machines*. IEEE Computer 30(9). pag. 86-93. Sept. 1997.