

A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification

Stephan Jourdan, Ronny Ronen, Michael Bekerman,
Bishara Shomar, and Adi Yoaz

Intel Corporation

Intel Israel (74) Ltd., IPA Dept., MS: IDC-3C
P.O. Box 1659, Haifa 31015, Israel

{sjourdan,ronen,bekerman,bshomar,ayoaz}@iil.intel.com

Abstract

Hardware renaming schemes provide multiple physical locations (register or memory) for each logical name. In current renaming schemes, a new physical location is allocated for each dispatched instruction regardless of its result value. However, these values exhibit a high level of temporal locality (result redundancy). This paper proposes:

1. **Physical Register Reuse.** To reuse a physical location whenever it is detected that an incoming result value matches a previous one. This is performed during register renaming and requires some VALUE-IDENTITY DETECTION hardware. By mapping several logical registers holding the same value to the same physical register, Physical Register Reuse gives the opportunities:

- **SHARING** – exploit the high level of value-redundancy in the register file to either reduce the file size and complexity, or effectively enlarge the active instruction window. Our results suggest reduction factors of 2 to 4 in some cases. Performance is increased either by the enlarged instruction window or by the higher frequency enabled by a smaller register file requiring fewer ports.
- **RESULT REUSE & DEPENDENCY REDIRECTION** - move the responsibility of generating results:
 - (1) From the functional units to the register renamer, resulting in the possible elimination of processed instructions from the execution stream.
 - (2) From one instruction to an earlier one in the instruction stream, possibly allowing dependent instructions to be scheduled earlier.

This way, large performance speedups are achieved.

2. **Unification.** To combine the memory renamer with the register renamer in order to extend the above-stated sharing and result reuse & dependency redirection ideas to both registers and memory locations. This allows even greater hardware savings and performance improvements. This also simplifies the processing of store instructions.

Keywords

Register and memory renaming, physical register reuse, value temporal locality, result reuse, dependency redirection.

1 Introduction

Most instructions operate on several source operands and generate results. They name, either explicitly or through an

indirection, the source and the destination locations where values are read from or written to. A **name** may be either a logical register or a location in memory.

A true dependency between two instructions is created whenever the second instruction, in program order, reads a source operand that is written by the first instruction. The second instruction is said to be register or memory dependent on the first instruction, and it cannot start execution before the first instruction has produced its results (neither parallel nor out-of-order execution). Dependencies are tracked to ensure program execution correctness. The register and memory **dependency tracking hardware** take into account the order of the instructions as well as the source and destination operand names:

- **REGISTER-DEPENDENCY tracking (scoreboarding)** is usually performed in the early stages of the pipeline where instructions are currently processed in order. The hardware records the last producer for each logical register, if any (register alias table). Memory operations are also tracked for register dependencies in addition to memory dependencies, to perform address computations and to collect the source data to be stored.
- **MEMORY-DEPENDENCIES** are more difficult to track because names (addresses) are known only after address computation, i.e. out of program order. Several studies have recently dealt with this problem by making memory operations dependent on other operations with which they are likely to collide. This is done by introducing false register-dependencies while dispatching potentially colliding memory operations, forcing their addresses to be generated in-order. After the address computation step, memory dependencies are tracked to ensure program correctness. The huge number of memory names prevents the use of an indexed table (alias table) to record in-flight computations. A buffer is required to hold all pending store addresses (memory ordering buffer). Memory-dependency tracking is often referred to as *memory disambiguation*.

Dependency tracking allows out-of-order processors to base their scheduling on register and memory dependencies only. However, anti-dependencies and output dependencies are introduced (false dependencies). To overcome this problem, current processors feature **renaming hardware** (both register and memory renamers). Renamers provide each instruction with a unique physical location into which it stores its result. *The register and memory renamers provide multiple storage*

locations for each logical name whereas the dependency-tracking hardware ensures that each instruction refers to the right copy. Typical physical locations are physical registers and entries in the memory forwarding buffer. Coupling the dependency-tracking hardware with the renamers handles all dependencies. Typically, they are all processed together since they share most resources and data-paths, e.g. the memory buffer holds both ordering and forwarding information while the register alias table maintains pointers to physical registers that were previously allocated, rather than instruction tags.

This paper focuses on advanced renaming techniques. It does not cover new dependency-tracking schemes. The main innovative renaming ideas are (1) to reuse a physical location whenever a result is detected to match a previous one by **Value-Identity Detection** hardware (**Physical Register Reuse**) and (2) to use the same physical locations for both the memory and the register renamer (**Unification**). These ideas make possible such novel techniques as **Sharing** and **Result Reuse/ Dependency Redirection** on all register and memory physical locations, boosting performance significantly.

The remainder of this paper is organized as follows. Subsections 1.1 and 1.2 describe the prior work and the simulation methodology. Section 2 focuses on register renaming techniques and introduces the *Physical Register Reuse* scheme. Several *Value-Identity Detection* schemes are discussed with a particular emphasis on *move elimination*. Section 3 introduces *unified renaming* to extend the techniques previously described to the memory renamer. Section 4 presents various recovery schemes. Some concluding remarks are provided in section 5.

1.1 Prior Work

This sub-section describes previous works on renaming techniques. Only the major milestones are listed.

The concept of **out-of-order execution** was first implemented in the IBM 360/90 [Ande67]. Later studies in the early 70s formalized the concept of out-of-order execution as in [Tjad70][Kell75]. The concept was revisited and extended in the 80s [Weiss84][Patt85]. None of these studies tackled the problem of exceptions.

[Smit85] first presented hardware schemes to manage exceptions precisely. Two major renaming structures were introduced: the reorder buffer and the history buffer. The **reorder buffer** provides physical locations to store the result of each instruction. Results update the processor state in this scheme once the associated instructions become *safe*, i.e. free of exceptions. The step in which exceptions are tracked is called *retirement* and it is processed in program order. The **history buffer** records the value overwritten by each update to the state at execution. The processor state can be easily repaired by writing back the original contents serially. However, restoring an architectural state using a reorder buffer or a history buffer upon an exception requires multiple cycles¹. While this is not a severe problem for regular exceptions (since they are uncommon), this significantly impairs performance on each branch misprediction in processors implementing speculative

¹ although a costly prioritized CAM-lookup reorder buffer is capable of responding instantaneously.

execution [Butl93]. [Hwu87] introduced the **checkpoint** repair scheme to deal efficiently with branches. Snapshots of the architectural register state are made when dispatching branches. Immediate recovery is done by restoring the content of the corresponding checkpoint. A more cost-effective scheme was derived and introduced in [Butl90].

Other studies in the late 80s and in the 90s dealt with dependency tracking and renaming techniques. Only implementation details and few improvements were discussed, as in [Moud93][Fark95]. A significant change is the virtual register renaming scheme [Gonz98] where the renaming is decoupled from the dependency tracking hardware and processed right before the write back stage.

Memory renamers traditionally feature a reorder buffer-like structure called a forwarding buffer to hold pending cache updates. The memory-dependency tracking hardware (memory ordering buffer) buffers at least all pending store addresses to determine true memory-dependencies. As explained in the introduction, the tracking cannot be performed *as is* since names are generated out-of-order. Early studies, as in [John91], focused on two schemes to overcome this tracking problem. The first scheme is to insert false register dependencies to force a strict in-order address-generation process (although loads are not made dependent on older loads). The second alternative is to let addresses be computed out-of-order and to recover when an error occurs, i.e. a true memory-dependency which was not detected because the addresses of the load and the prior store were computed in reverse program order. Tracking errors are discovered by comparing newly-generated store addresses to all younger load addresses that have been already computed. [Gall94] and [Fran96] discussed implementations of such schemes.

Recently, [Mosh97a] proposed to predict the occurrences of such tracking errors based on past history. Such prediction schemes reduce the number of false dependencies while minimizing the number of tracking errors. [Mosh97a] like [Chry98] also proposed to predict all memory true-dependencies regardless of the address computation ordering. However, this scheme typically requires more predictor space since the number of true memory-dependencies is far higher than the number of tracking errors. On the other hand, predicting the true memory dependencies allows for speculative forwardings from stores to loads through the memory renamer entries as described in [Mosh97b] and [Tyso97].

Finally, the Speculative Memory Forwarding scheme introduced in [Mosh97b] proposed to bypass the memory renamer by processing *speculative* forwardings directly in the register renamer.

1.2 Simulation Methodology

Results provided in this paper were collected from an IA-32 trace-driven simulator. Each trace consists of 30 million consecutive IA-32 instructions translated into micro-instructions on which the simulator works. Traces are representative of the entire execution. Results are reported for 47 traces grouped into 8 suites:

- SpecInt95 (**INT** - 8 traces),
- CAD programs (**CAD** - 2),

- multimedia programs using MMX™ instructions (**MM** - 8),
- games, e.g. Quake (**GAM** - 5),
- programs written in JAVA (**JAV** - 5),
- some TPC benchmarks (**TPC** - 3),
- common programs running on NT, e.g. Word, Excel (**NT** - 8),
- and common programs running on Windows 95 (**W95** - 8).

In this paper, we focus more on statistical results which emphasize the motivations for the various suggestions. Performance results are only briefly presented, mainly to give a flavor of the potential benefit. Performance results are deemphasized, as actual performance benefits are highly dependent on the implementation and may vary a lot. Choosing an arbitrary configuration (whether current or futuristic) may give biased results of questionable significance.

2 Advanced Register Renaming

2.1 Current Register Dependency-Tracking and Renaming Techniques

Modern processors exploit out of order execution to speed up processing time. Out of Order execution involves a mechanism called register renaming in which the processor maps logical registers into physical locations. Register renaming is used to remove register anti-dependencies and output-dependencies and to recover from control speculation. The basic register renaming mechanism is well known and widely used (e.g. Intel® Pentium® Pro Processor [Inte96]). This section presents the most advanced combined register renaming and dependency-tracking scheme involving three structures: a Free List (FL), a Register Alias Table (RAT), and an Active List (AL). This scheme has been used in the MIPS R10000 and DEC 21264.

The RAT maintains the latest **mapping**² for each logical register. The RAT is indexed by the source logical registers, and provides the mappings to the corresponding physical registers (dependency-tracking). For each logical destination register specified by the renamed instructions, the allocator (renamer) provides an unused physical register from FL. The RAT is updated with these new mappings. Physical registers can be reclaimed once they cannot be referenced anymore. Once a logical register is renamed, all subsequent instructions can only access the new mapping; i.e. they cannot read the physical register previously mapped. Thus, an appropriate and straightforward condition for register reclaiming is to reclaim a physical register only when the instruction that evicted it from the RAT retires. As a result, whenever a new mapping updates the RAT, the evicted old mapping is pushed into AL (an AL entry is provided to each instruction). When an instruction retires, the physical register of the old mapping recorded in AL, if any, is reclaimed and pushed into FL. This cycle is depicted in figure 1.

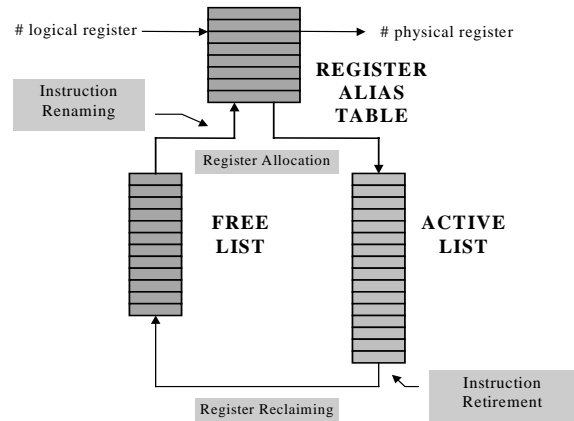


Figure 1. Register Renaming.

2.2 Physical Register Reuse

Motivation. Most instructions operate on several source operands and generate results. These results are recorded into local physical registers allocated for each instruction, so that dependent instructions can operate on them. The range of generated values is usually limited. Indeed, integer results are often pretty small and the same value may be generated several times by different instructions currently in the instruction window. A perfect example is Boolean values such as control-flow conditions. Figure 2 shows the percentage of computed values that match one of the values generated by preceding instructions according to the number of prior instructions scanned (16, 32, 64, 128, or 256 instructions). Note that the scanned values were not filtered for duplicate results so they may also exhibit a high-level of redundancy. Results are highlighted for four of the SpecInt95 benchmarks, and confirm our claim for programs compiled to run on an IA-32 processor.

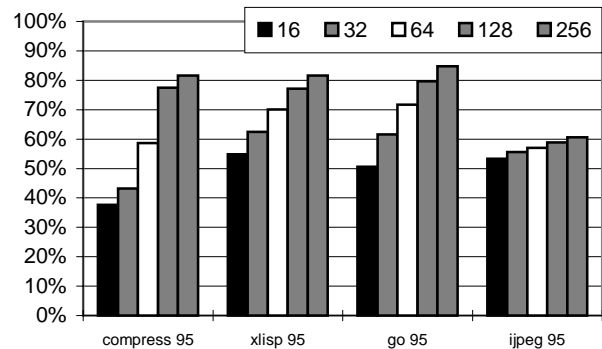


Figure 2. Number of Identical Results.

Concept. Physical registers hold values that are part of the architectural states currently alive in the machine. A physical register is allocated for every result regardless of its value. However, there is no reason to allocate separate physical registers when they maintain the same value. This paper proposes to reuse a physical register whenever we *detect* that an incoming result value matches a previous one. Physical Register Reuse relies on a Value-Identity Detection hardware to perform the detection prior to register renaming. The detector outcome can be either *safe* or *speculative*. By mapping several logical

² Throughout the paper, we refer to a mapping as the pairing of a logical register with a physical register it maps to.

registers holding the same value to the same physical register, Physical Register Reuse allows:

- **SHARING** - exploit the high level of value-redundancy in the register file to either reduce its size and complexity or effectively enlarge the active instruction window. This is critical since register file size and complexity depend on both the depth of the instruction window and the width of the processor, parameters that are constantly increasing. Performance can be increased either by an enlarged instruction window or by higher frequency enabled by a smaller register file featuring fewer ports.
- **RESULT REUSE & DEPENDENCY REDIRECTION** - move the responsibility of generating results:

(1) From the functional units to the register renamer (elimination property). When the identity detection is speculative, the functional units must verify the prediction. When the detection is safe, the instruction is fully eliminated from the execution stream since it is processed by the renamer.

(2) From one instruction to an earlier one in the instruction stream (redirection property). The two instructions can be unrelated and the earlier instruction may have not been scheduled yet. Instructions that are true-dependent on the incoming renamed instruction, are now linked to the earlier instruction. This most probably results in an early execution of these dependent instructions. **True register dependencies are dynamically redirected to shorten critical paths.** By doing so, the core ends up processing an equivalent data-flow graph more suitable for fast execution. Note that when the two instructions belong to the same dependency path, the data-flow graph is surely collapsed.

Section 2.3 provides an extensive description of the move elimination technique which makes use of all the above-mentioned concepts, while using a straightforward Value-Identity Detector.

Multiple Register Mappings. Existing renaming schemes are not capable of supporting Physical Register Reuse, i.e. in all current register renaming schemes, a physical register is mapped by at most one logical register. To allow several mappings to the same physical register, we propose to extend the current register renaming scheme detailed in section 2.1 by adding a counter to each physical register. Counter values represent the number of mappings to each physical register. When a physical register is allocated, its counter value is incremented. The counter is decremented when the register is reclaimed. The use of a counter ensures that no physical register is lost during the cycle depicted in figure 1. In the traditional register renaming scheme, such a scheme means that registers in FL have their counters set to 0 while registers in AL or in the RAT have a value of 1. In the proposed scheme, whenever a physical register is reallocated, its counter value is incremented, and the physical register becomes free again only when it has been reclaimed the same number of times it has been allocated. The maximum number of concurrent allocations for a given physical register depends on the number of bits allocated to the counters.

Example: Let's assume the following instruction sequence where instruction (b) is detected to produce the same value as instruction (a) (new mappings are bolded).

	Instruction	eax mapping	ebx mapping
(a)	eax <-	pr39	pr13
...			
(b)	ebx <-	pr39	pr39
...			
(c)	eax <-	pr53	pr39
...			
(d)	ebx <-	pr53	pr21

When dispatching (a), *eax* is mapped to a new physical register, e.g. *pr39*. Its counter is set to 1. When processing (b), the renamer reuses *pr39*. *pr39*'s counter is incremented to 2. Instructions between (b) and (c) may read either *eax* or *ebx* and access *pr39* as a result. The true dependencies of all instructions dependent on (b) are **redirected** to (a), i.e. these instructions become dependent on (a) rather than (b). (c) kills the old mapping of *eax* and maps *eax* to a new physical register. *pr39*'s counter is decremented to 1 when (c) retires. However, *ebx* is still alive and subsequent instructions should be able to access its value. Deliberately, *pr39* is not freed since its counter is not reset. It is freed only when (d) retires. This is safe since none of the subsequent instructions can access it.

Value-Identity Detection. We have already shown in figure 2 the high-level of value temporal locality. In order to exploit the potential of Physical Register Reuse, Value-Identity Detection schemes are used to determine which physical register can be reused, if any. Their outcomes can be either safe or speculative. The latter case requires new recovery mechanisms that are discussed in section 4. The remainder of this sub-section describes general Value-Identity Detectors that are based on:

- **VALUE MATCHING.** The idea is to cache the last few generated values and to perform a lookup with the instruction results. A very high hit-rate can be obtained from a small cache according to the results shown in figure 2 (expect 70-75% on average with a 32-entry cache³). The cache can be as big as the physical register file (no value cache in this extreme case but some extra CAM ports in the file). The lookup must be performed prior to register renaming. As a result, it can be done at the output of:

1. The functional units if register renaming is performed right before the write-back stage like in the *virtual renaming* scheme [Gonz98]⁴. The value-cache lookup is done while writing back the results into the file. A cache hit leads to the immediate reclaiming of the newly allocated register. This scheme is *safe*.

³ The cache only records non-identical results. This was not the case in figure 2.

⁴ Virtual renaming requires a mechanism to prevent deadlocks due to a lack of physical registers (the register allocation is done out of program order). [Gonz98] introduces techniques to anticipate such a problem. An alternate, presumably cleaner, scheme is to steal the physical register already allocated to a younger instruction which now needs to be re-scheduled. Such a scheme does not rely on any data recovery mechanism (like for value-prediction) if the younger instruction is chosen in a way that none of its dependent instructions were scheduled.

2. The *value predictor* [Lipa96] with the predicted value as the index, in the front-end pipeline. A match on a value-cache entry with the predicted value leads to the reuse of the associated physical register. The value prediction scheme remains unchanged otherwise. This scheme is *speculative*. Note that no writing is performed into the register file on a match. This gives the opportunity to remove all the costly extra write-ports required with value prediction, at the expense of a reduction in the number of predictions performed.
 3. The *reuse buffer* [Soda97] if the reused value is still in a physical register. This scheme is *safe*.
 4. The *instruction decoder* when the instruction result can be statically derived from the instruction itself. A simple example is immediate-to-register moves as explained in section 2.3. This scheme is also *safe*.
- **DIRECT IDENTITY DETECTION.** The schemes described above perform the identity detection by comparing values. An alternative option is to detect the identity property directly, without even knowing the values. We distinguish two classes based on the safety of the outcomes:

1. *Safe.* The identity is known *a priori*. Move instructions are the typical target of such a class (detailed in section 2.3) but a more advanced scheme may detect the identity over *sequences of instructions* (e.g. a simple pattern like an increment instruction followed by a decrement instruction). Note that a dependency path does not need to link the two instructions. Indeed, the detector can base its outcome on the definition of pending operations and on the knowledge of currently generated results.
2. *Speculative.* The idea is to predict the identity rather than the value itself, i.e. an incoming instruction is predicted to produce the same value as an earlier instruction in the instruction stream, though the actual value is not known (the earlier result is reused). Direct applications are the elimination of conditional moves from the executed instruction stream and the collapsing of true memory-dependencies as explained in section 3.

Sharing. All these schemes reduce the number of physical registers and write ports into the register file. With one exception, the reduction is partial since the schemes only tackle a portion of the instruction stream. The enhanced *virtual-renaming* scheme performs value-cache lookups on all computed results, suggesting a potential reduction factor of 2 to 4 with a 32-entry value cache depending on the temporal locality of the result values. Note that this gain is much bigger than whatever has been reported in previous studies where allocation timing was the major issue (e.g. [Gonz98]). We do not further elaborate on this sharing property since a complete study is beyond the scope of this paper.

Result Reuse & Dependency Redirection. Most of the schemes performs result reuse (enhanced *virtual-renaming* is again an exception since results have already been computed). As a result, true dependencies on processed instructions are redirected. The safe schemes fully eliminate instructions from the execution stream, while with the speculative schemes, the functional units still need to perform the verification of the

identity predictions. Note that some of the schemes already perform the dependency collapsing (*value prediction* and *instruction reuse*). They do not benefit from result reuse.

2.3 Case Study: Move Elimination

This sub-section covers a straightforward example of value-identity detector. Its simplicity should not blur the potential behind the *result reuse/dependency redirection* concept.

Motivation. In most IA-32 instructions, one of the source operands is also used as the destination operand. This means that the original value is lost and cannot be used by subsequent instructions. If the lost value is needed by later instructions, a *move* instruction is inserted prior to the destructive instruction. This *move* instruction copies the original value into another logical location so it can be accessed by instructions following the destructive instruction. Another reason for the insertion of *move* instructions is to set the parameter values in the right registers prior to a procedure call⁵. As a result, the number of *move* instructions is quite significant in typical IA-32 programs. Figure 3 highlights their number in common programs. Note that *move* micro-instructions only operate on registers (either integer or floating-point registers). *Move* instructions operating on memory operands are translated into load or store micro-instructions (discussed in section 3). In RISC ISAs, move instructions are far less common and mainly used for parameter passing.

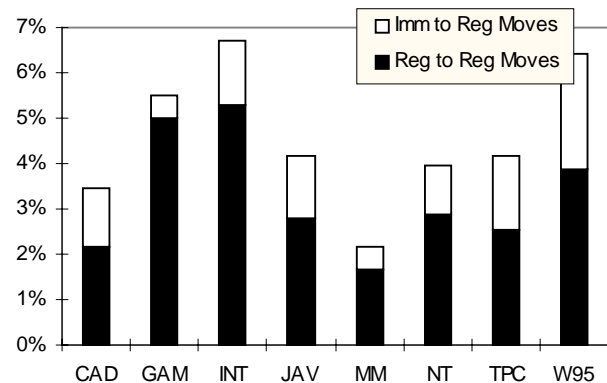


Figure 3. Dynamic Number of Move Micro-Instructions as a Fraction of All Micro-Instructions.

Move Elimination. The move elimination technique makes use of the Physical Register Reuse scheme and all the related concepts introduced in section 2.2. We distinguish two sets of move instructions:

- **REGISTER-TO-REGISTER** moves are a straightforward example for safe reuse of physical registers using the Direct Value Identity Detection scheme. The execution of such operations can simply be done by just mapping the destination logical register to the physical register mapped by the source logical register. This is done by allocating the source physical register again and by incrementing its counter. Such move

⁵ IA-32 Application Binary Interface (ABI) requires that parameters are passed on the stack. However, compilers do use alternate, non standard, register-based parameter passing, when possible.

instructions that are processed in the renaming stage are not dispatched for execution. They remain in the instruction window ready to be retired (result reuse - elimination property). This technique collapses the dependency graph (result reuse - redirection property) since instructions dependent on the move are made dependent on the real producer instead (dependencies are redirected). It also reduces the requirement for physical registers and fewer writes are performed into the register file (sharing). Potential performance improvements due to the elimination of register-to-register moves are shown in section 3.2.

- IMMEDIATE-TO-REGISTER moves can be safely eliminated using the Value-Matching detection scheme as explained in section 2.2. The value-cache lookup is performed at the output of the decoder with the immediate value encoded in the opcode. On a hit, the move is eliminated as described for register-to-register moves. Hence, boosting performance.

3 Advanced Memory Renaming

3.1 Current Memory Dependency-Tracking and Renaming Techniques

Memory renaming is usually performed in a reorder buffer-like structure called a forwarding buffer where store instructions write into instead of updating the memory hierarchy. Upon a store retirement, cache updates are processed. Stores can be seen as move instructions from the register files (or from an immediate value) to the forwarding buffer. Whenever a true memory dependency arises, the colliding load gets its value from the forwarding buffer rather than from the cache.

Dependency tracking is done by recording, at least, all pending store addresses in an ordering buffer. The ordering buffer and the forwarding buffer are often combined. If addresses are not forced to be computed in-order (the disambiguation process is not safe), load addresses must also be recorded in order to check newly generated store addresses against younger already-computed load addresses. Maintaining total memory order between all memory references may result in a large full-scale CAM reorder buffer. Cost-effective schemes can keep load addresses separate from store addresses reducing the complexity while still maintaining global memory ordering.

By definition, store instructions depend both on the source data to be stored in memory and on the address operands. Executing stores involves moving values into the memory renamer locations while computing their memory addresses. Processing both operations simultaneously may introduce unnecessary delays:

- Forcing the move to wait for the address computation slows its execution when the source data is ready before the address. In reality, this is an uncommon event.
- A real concern that may impair performance is the other way around. The address is usually computed before the data is ready. Forcing the address to wait for the data stalls all loads which are false register-dependent on this store.

These two concerns suggest splitting store instructions into two independent micro-instructions, i.e. a store data and a store address. However, predicting memory dependencies alleviates

the performance loss due to stalled loads that are falsely register-dependent on stores they do not collide with: [Mosh97a] reported that prediction schemes manage to reduce significantly the number of such loads. Therefore, a dependency predictor should eliminate the need for the decoupling of the address computation from the data moving.

3.2 Unified Renaming

Motivation. Register spills are responsible for most of the true memory-dependencies. Their number depends mostly on the number of logical registers provided by the ISA. For instance, an IA-32 program features many more spills than the same program compiled for a traditional RISC processor. In IA-32 programs, other noticeable causes for true memory dependencies are parameter-passing and return address push/pop⁶ on a procedure call/return. [Mosh97a] showed that the number of memory dependencies is high even in RISC ISAs. Furthermore, the number of memory dependencies increases quite significantly when deepening the instruction window. Figure 4 shows the percentage of dynamic loads that collide with preceding stores as a function of the number of prior instructions scanned (32, 64, 128, or 256 instructions). Based on these results, handling colliding loads efficiently should improve performance significantly. Processing these colliding loads through a forwarding buffer has been shown not to be optimal [Mosh97b]. Furthermore, a forwarding buffer complicates the execution of store instructions since it requires a value transfer among other things. Finally, part of the value redundancy highlighted in figure 2 comes from values loaded from memory.

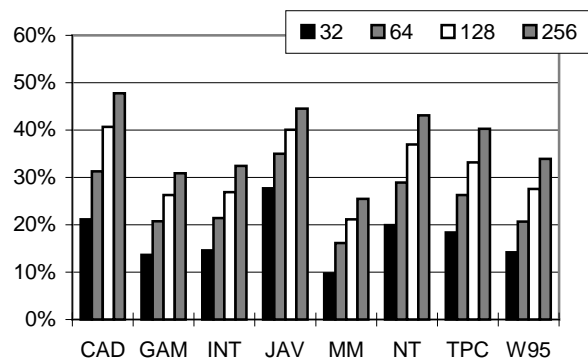


Figure 4. Number of True Memory-Dependencies as a Fraction of all Load Micro-Instructions.

Concept. The main idea is to use the same physical locations (physical registers) for both the register renamer and the memory renamer. Hence the term *unified renaming*. Memory dependencies are predicted in the front-end pipeline, thereby enabling us to apply all the schemes described in section 2.2 to the memory renamer in order to overcome the problems mentioned above:

- **Sharing.** The sharing property of the Physical Register Reuse scheme deals with any *value redundancy* in a unified renamer. Furthermore, since all values in the memory renamer were duplicated from the register renamer, *unifying*

⁶ In IA-32, the return address is recorded into the stack.

does not increase the number of physical registers required. We do not further elaborate on this sharing property.

- **Memory-Dependency Collapsing.** A true memory dependency is nothing more than a register move from an instruction on which the store depends to the load: this instruction result is reused by the load instruction (redirection property). The Result Reuse technique can collapse memory dependencies to boost performance as explained in section 2.3 for move instructions. Furthermore, the Result Reuse technique can partially eliminate the load instruction (elimination property) since the cache access is not required for verification.
- **Store Processing.** Stores are processed more efficiently since they collect their source data directly from the register file, i.e. the store-data operation that moved the value to be stored from the register renamer to the memory renamer, is eliminated. Furthermore the Physical Register Reuse scheme means that no store buffer⁷ is required.

Note that memory-dependency tracking still needs to be performed in an ordering buffer since the detector outcome is speculative and must be verified (see the result reuse paragraph in section 2.2).

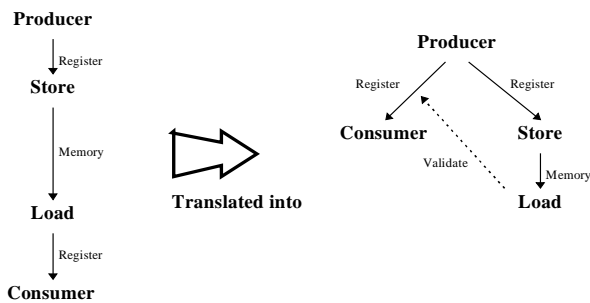


Figure 5. Memory-Dependency Collapsing [Mosh97b].

The Speculative Memory Forwarding scheme introduced in [Mosh97b] proposed to bypass the memory renamer to collapse *predicted* true memory dependencies: the destination logical register of a colliding load is mapped to the source data physical register of the corresponding store. The verification of the prediction is done through memory, and ensures that the loaded value matches the speculative one. Figure 5 summarizes this dynamic translation of the dependency graph done during the register renaming process, highlighting the communication channels for all dependencies, i.e. registers or main memory.

Unified renaming is a significant extension of the Speculative Memory Forwarding scheme as shown later in this section since:

- Speculative Memory Forwarding only performs the collapsing. Unified renaming also tackles value-redundancy in all physical locations (register and memory renamers).
- Dependent loads no longer access the memory hierarchy. Verification is done by comparing addresses (elimination property).

- By means of the Multiple Mapping scheme previously introduced, memory-dependency collapsing can be performed under the most advanced register renaming techniques described in section 2.1.
- Unified renaming removes both the forwarding buffer and the store buffer.
- Unified renaming not only improves the processing of loads but it also handles store instructions in a more efficient way.

Memory-Dependency Collapsing. Memory operations move data from/to the register file. They perform no computation but they describe a dependency arc between the producer of the value to be stored and the consumers of the loaded value. Such dependencies are not limited to a small number of logical locations, they can extend well beyond the size of the instruction window, and they are fully dynamic because of the way names (addresses) are generated. Unified renaming relies on a dependency predictor to detect true memory-dependencies inside the instruction window. Since memory dependencies are nothing more than regular register moves, the underlying idea is to bypass memory by pre-processing the move in the register exactly like the move elimination technique discussed in section 2.3, i.e. the memory dependency arc is translated into register dependency arcs linking the producer of the value stored in memory and the consumers of the load - the producer result is speculatively reused by the load. This significantly collapses the dependency graph and reduces execution time since memory dependencies consume many cycles in synchronization and in value transfer in schemes featuring at best a forwarding buffer. Besides the expected increase in performance, some hardware savings are achieved by using the physical register file instead of a costly forwarding buffer to handle true memory-dependencies.

For each predicted collision, the memory-dependency predictor provides information about the location of the collided store in the instruction window. The physical register holding the value to be stored can be retrieved easily. The dependency predictor acts as a value-identity detector. During renaming, the destination logical register of the predicted colliding load is mapped to this physical register: the physical register is reallocated, i.e. its counter is incremented.

Instructions dependent on a predicted colliding load (consumers) get their source operands mapped to the reallocated physical register. These consumers can execute back-to-back with the producer of the stored value. Memory is completely bypassed and the data-flow graph is collapsed since the dependency redirection is performed on instructions belonging to the same dependency path as explained in section 2.2. Furthermore, on the contrary of what has been described in [Mosh97b], the colliding load does not need to access the cache since the result of the cache access is not used at all⁸. Colliding loads just compute their addresses to verify the predicted match with the corresponding store address. This reduces the load memory traffic (32% on average in a 128-deep processor as shown in figure 4) and achieves better performance. One may argue that this validation scheme is conservative since, in many

⁷ The store buffer is the structure used to record pending cache updates associated with retired store instructions.

⁸ There is no memory coherency problem (multiprocessor mode) since the ordering buffer is snooped for a match along with the on-chip data caches.

cases, the values are identical even if the addresses differ. Indeed, we are performing a value-identity prediction and not a memory-dependency prediction (another difference with Speculative Memory Forwarding). This suggests a two-level verification methodology where the first step processes the address verification. Only on an address mismatch, the cache access is performed to verify the value. The re-scheduling of all dependent instructions is therefore done only on a value mismatch. This should definitely boost performance beyond the results reported below, while still significantly saving memory bandwidth.

The prediction verification is done in the ordering buffer, and it must ensure that both the load and the store access the same memory location and that no other store in between writes to this location. Furthermore, the store may retire before the load address computation. To solve this particular case, the store address is forwarded to the load entry in the ordering buffer and later verified much like in a load-address prediction scheme. The disambiguation process resolves all the remaining issues including the checking that no store in between writes to the same location.

Figure 6 show performance improvement due to register-to-register move elimination, better memory disambiguation, and better memory renaming techniques. Results are shown only to give a flavor of the potential benefit. All speedups are over a *baseline* configuration where addresses are computed in-order with respect to older store addresses. *Disambiguation* refers to a scheme that inserts register-dependencies only on true memory-dependencies, while *unified* is the scheme described in this section. Both memory schemes feature a perfect dependency predictor. The machine configuration is a 8-wide 128-deep out-of-order processor featuring 8 uniform functional units and a 32KB first-level data cache. Instruction latencies are common to Intel's processors. Results show that unified renaming boosts performance by 27% on average over *baseline*, and 10% over *disambiguation*. The *register-to-register move elimination* scheme alone increases performance by 1.5% on average, and by 29% when combined with *unified renaming*. The unusually large speedup reported for JAVA applications can be explained by the large number of memory operations. This is due to the stack-based model and short procedures used in JAVA bytecode, and to the lack of optimizations performed by JAVA JIT compilers.

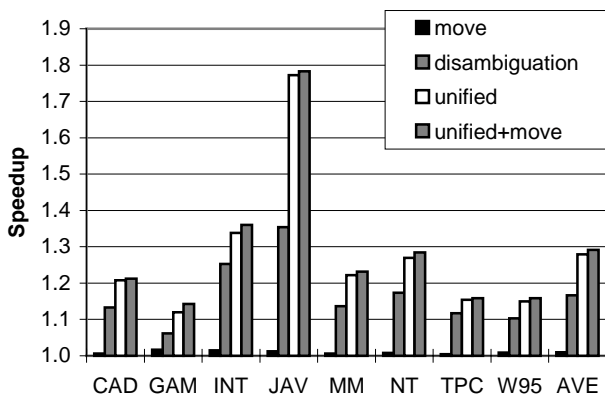


Figure 6. Performance Improvement.

Finally, the memory cloaking scheme introduced in [Mosh97b] proposed to speed up the forwarding process between stores and loads even if they do not fit in the instruction window (mainly do not wait for addresses to be computed). We claim that a load-address prediction scheme that schedules cache-accesses early in the pipeline, performs the dependency collapsing better since it is not limited to colliding load-store pairs. We therefore do not extend the value-identity detector to take advantage of reclaimed physical registers that still hold values that were stored in memory.

Example: Let's assume the following instruction sequence where the load (c) collides with the store (b).

	Instruction	eax mapping	ebx mapping
(a)	eax <-	pr39	pr13
...			
(b)	[A] <- eax	pr39	pr13
...			
(c)	ebx <- [A]	pr39	pr39
...			
(d)	<- ebx	pr39	pr39

When dispatching (a), *eax* is mapped to a new physical register, let's say *pr39*. Its counter is set to 1. (c) is renamed and predicted to be memory-dependent on (b): *pr39* is reused and its counter is incremented. This means that (d) reads directly from *pr39* and that it may execute back-to-back with (a), completely bypassing memory.

Store Processing. In unified renaming, store instructions no longer move the source data out of the register file at the execution stage since the memory renamer works on physical registers (elimination of the store data micro-instruction). There is also no need to move the value to a store buffer entry at retirement. Instead, the register file is read when updating the cache, which can be many cycles after retirement⁹. However, subsequent retiring instructions may reclaim the physical register maintaining the value that still needs to be written in the cache. To overcome this problem, the counter associated with this physical register is incremented when the store is renamed. It is decremented when the cache update is performed. The RAT is not updated since no logical register is associated.

Stores may specify an immediate value as source data. A new physical register should be allocated to hold temporarily this immediate value, unless a register can be reused as explained in section 2.3 for immediate-to-register moves. In any case, a newly allocated register is freed when the cache update is performed as mentioned above.

⁹ Note that we do not introduce any requirement for extra read ports. Indeed, the reading can be performed with the port associated to the former store-data micro-instruction. Furthermore, delaying the reclaiming of the register is not expected to impair performance significantly since the value is likely to be re-used by younger loads or any other instruction.

4 Recovery Schemes

Part of the schemes introduced above perform Physical Register Reusing by predicting identity relations. On a misprediction, the execution core must be repaired. Unlike control mis-speculations, the validity of the instructions is not questioned. However, all the instructions belonging to the sub-graph starting from the faulting instructions misrepresent the data-flow graph.

On a misprediction, we cannot just re-execute the faulting instruction as proposed in [Tyso97]. The faulting instruction needs a different physical register, which must be broadcasted to all dependent instructions¹⁰. However, all dependency arcs are marked by means of physical registers in the wake-up logic of the instruction scheduler. Instructions dependent on the faulting one cannot be distinguished from instructions that are dependent on other producers of the wrongly reused physical register. We present two ways to overcome this problem:

- All instructions past the faulting one are re-renamed. This requires repairing all renaming structures. In other words, backtrack to the closest previous checkpoint. Note that instructions just need to be re-renamed and not re-fetched.
- Instructions record pointers to the instructions they depend on in addition to the source physical registers, much like the tags used in virtual renaming [Gonz98]. On a misprediction, the new physical register is broadcasted with the associated tag as an index to update the source operand of the first-level dependent instructions only. Next, the faulting instruction is re-executed. This results in a cascade re-execution of all truly dependent instructions. A lack of physical registers can be solved as explained in section 2.2 with the *stealing* scheme.

Example: Let's consider the following instruction sequence where (c) has been wrongly predicted colliding with (b):

	Instruction	dependent on
(a)	eax <- 3	
(b)	[A] <- eax	(a)
...	(none of these instructions rename eax)	
(c)	ebx <- [B]	
(d)	ecx <- ecx + ebx	(c)
(e)	edx <- edx + eax	(a)
(f)	ebx <- ebx + edx	(c,e)
(g)	edx <- edx * ecx	(d,e)

In the first scheme, (c) and all subsequent instructions are sent back to the renamer. (c) gets a new physical register from FL and accesses the cache.

¹⁰ Only the instructions directly dependent on the faulting one needs to be updated.

In the second scheme, the tags help distinguish the independent instruction (e) from all the instructions that really depend on the load (c). A new physical register is allocated to (c) from FL and it is broadcasted with (c) as an index. Instructions (d) and (f) are updated. (c) is re-executed, accessing the cache, and writes back into the newly allocated physical register. As a result, instructions (d), (f), and, later, (g) are re-executed.

5 Concluding Remarks

Summary. In this paper, we highlighted the high level of *value temporal locality*. We introduced the ideas of reusing physical registers (*Physical Register Reuse*) and combining the memory renamer with the register renamer (*Unification*) to exploit this locality. These ideas enable the implementation of powerful techniques such as *Result Reuse/ Dependency Redirection* and *Sharing* in all physical locations (register and memory). Higher processor performance is achieved while saving some hardware, as illustrated by the move elimination and unified renaming examples. We described modifications to state-of-the-art renaming schemes, these modifications accommodate Physical Register Reuse and Unified Renaming. *Value-Identity Detection* schemes determine which physical register can be reused, if any. These detection schemes were grouped into two classes depending on whether the identity is directly detected or indirectly by comparing values. We introduced several schemes for each class. In addition, the detector outcome can be either safe or speculative. The latter case requires appropriate verification and recovery schemes that were discussed.

Sharing. The sharing property of the Physical Register Reuse scheme seems very promising. Further work is required to determine the temporal stability of value-redundancy. If high reduction factors are reached, Physical Register Reuse is definitely an efficient scheme to implement large instruction windows without sacrificing clock frequency. This becomes critical since large windows become more and more attractive as new control-flow prediction mechanisms and related techniques (e.g. hardware predication) are introduced.

Result Reuse. Results are reused to eliminate instructions from the execution stream and to redirect true dependencies. **Dependency Redirection where the processor processes a translated equivalent data-flow graph, is a new paradigm for high performance.** The key issue here is the Value-Identity Detector. A direction for further work is to identify new relevant targets. Detectors can be split into two classes:

- *Safe.* Defining detectors working over sequences of instructions seems a promising challenge.
- *Speculative.* A formidable task is to locate correlation between independent instructions and to speculatively redirect the true dependencies to expose more parallelism.

Further Work. Other challenging research directions include:

- Building an advanced Value-Identity Detector for a very wide machine. For practical reasons, we may consider embedding the detector within a trace cache and performing some of the work semi-statically while building traces.

- Defining more efficient recovery schemes to dynamically update the data-flow graph represented in the machine.
- Studying how the schemes proposed in this paper affect various hardware configurations. This paper focused on concepts and ideas, and not on bottom-line, configuration-dependent performance numbers. Of particular interest are bounded configurations, where applying these schemes result in higher speedups that make the overall performance close to the original machine. For example, unified renaming should give higher speedups when bounding the memory-pipeline throughput while increasing the cache hit rate. This would enable some hardware savings in the memory pipelines without impairing performance significantly.

Architecture. This paper focused on micro-architectural techniques that benefit from the Physical Register Reuse scheme. With some modified architectural semantics, the usage of physical register reuse can be extended to include *undefined* registers. A register may be considered undefined before its first assignment, or when it is explicitly killed. Examples are the FINIT IA-32 instructions that clears the entire floating-point register stack, or on procedure returns when some of the registers that hold return values are not used (*dead* - see also [Mart97]). An undefined logical register can be mapped to any physical register since its value matches any other.

6 Acknowledgements

We would like to thank Dave Sager, Todd Austin, Bob Valentine, Andy Glew, Ilan Spillinger, and Gad Sheaffer (all presently in Intel Corporation) for their critical comments on early versions of the paper. We gratefully acknowledge the help and encouragement of the other members of our research group, in particular, Uri Weiser, Mattan Erez, and Lihu Rappoport. We also express our gratitude to Adina Hagege for her help in polishing the paper, and to Andre Seznec (presently in IRISA) and the anonymous reviewers for their insightful comments.

7 References

- [Ande67] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling", in IBM Journal of Research and Development, V.11 Jan. 67, pp.8-24.
- [Butl90] M. Butler and Y. N. Patt, "An Area-Efficient Register Alias Table for Implementing HPS", in Proceedings of the 1990 International Conference on Parallel Processing, 1990.
- [Butl93] M. Butler and Y. N. Patt, "A Comparative Performance Evaluation of Various State Maintenance Mechanisms", in Proceedings of the 26th Annual International Symposium on Microarchitecture, Austin, TX, December 1993, pp.70-79.
- [Chry98] G. Z. Chrysos and J. S. Emer, "Memory Dependence Prediction using Store Sets", in Proceedings of the 25th International Symposium on Computer Architecture, Barcelona, Spain, June 1998.
- [Fark95] K. I. Farkas, N. P. Jouppi, and P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors", in Digital WRL Research Report 95/10, November 1995.
- [Fran96] M. Franklin and G. S. Sohi, "A Hardware Mechanism for Dynamic Reordering of Memory References", in IEEE Transactions on Computers, V. C-45, NO.5, May 1996, pp.552-571.
- [Gall94] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer", in Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994, pp.183-193.
- [Gonz98] A. Gonzalez, J. Gonzalez, and M. Valero, "Virtual-Physical Registers", in Proceedings of the 4th International Symposium on High Performance Computer Architecture, February 1998.
- [Kell75] R. M. Keller, "Look-Ahead Processors", in Computing Surveys, V.7, NO.4, December 1975, pp.177-195.
- [Hwu87] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", in IEEE Transactions on Computers, V. C-36, NO.12, December 1987.
- [Inte96] Intel Corporation, Pentium® Pro Family Developer's Manual. Volume 2: Programmer's Reference Manual, 1996.
- [John91] Mike Johnson, Superscalar Microprocessor Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [Lipa96] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction", in Proceedings of the 29th Annual international Symposium on Microarchitecture, December 1996, pp.226-237.
- [Mart97] M. M. Martin, A. Roth and C.N. Fischer, "Exploiting Dead Value information", in Proceedings of the 30th Annual international Symposium on Microarchitecture, December 1997, pp.125-135.
- [Mosh97a] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Speculation and Synchronization of Data Dependencies", in Proceedings of the 24th International Symposium on Computer Architecture, June 1997.
- [Mosh97b] A. I. Moshovos and G. S. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction", in Proceedings of the 30th Annual international Symposium on Microarchitecture, December 1997, pp.235-245.
- [Moud93] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: an Alternative Approach", in Proceedings of the 26th Annual International Symposium on Microarchitecture, Austin, TX, December 1993, pp.202-213.
- [Patt85] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction", in Proceedings of the 18th Annual Workshop on Microprogramming, Pacific Grove, CA, December 1985, pp.103-118.
- [Smit85] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors", in Proceedings of the 12th International Symposium on Computer Architecture, 1985..
- [Smit95] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors", in Proceedings of the IEEE, Dec. 1995, pp1609-1624.
- [Soda97] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", in Proceedings of the 24th International Symposium on Computer Architecture, June 1997.
- [Tjad70] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions", in IEEE Transactions on Computers, V. C-19, NO.10, October 1970, pp.889-895.
- [Tyso97] G. S. Tyson and T. M. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming", in Proceedings of the 30th Annual international Symposium on Microarchitecture, December 1997, pp.218-227.
- [Weis84] S. Weiss and J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers", in IEEE Transactions on Computers, V. C-33, NO.11, November 1984, pp.1013-1022.