



# Intel<sup>®</sup> Technology Journal

Managed Runtime Technologies

## The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments

# The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments

Ali-Reza Adl-Tabatabai, Microprocessor Research Labs, Intel Corporation  
Jay Bharadwaj, Microprocessor Research Labs, Intel Corporation  
Dong-Yuan Chen, Microprocessor Research Labs, Intel Corporation  
Anwar Ghuloum, Microprocessor Research Labs, Intel Corporation  
Vijay Menon, Microprocessor Research Labs, Intel Corporation  
Brian Murphy, Microprocessor Research Labs, Intel Corporation  
Mauricio Serrano, Microprocessor Research Labs, Intel Corporation  
Tatiana Shpeisman, Microprocessor Research Labs, Intel Corporation

Index words: Just-in-time compiler, JIT, Java, Common Language Runtime, virtual machine, dynamic optimization

## ABSTRACT

Dynamic compilers (or Just-in-Time [JIT] compilers) are a key component of managed runtime environments. This paper describes the design and implementation of the StarJIT compiler, a dynamic compiler for Java Virtual Machines and Common Language Runtime platforms. The goal of the StarJIT compiler is to build an infrastructure to research the influence of managed runtime environments on Intel architectures. The StarJIT compiler can compile both Java\* and Common Language Infrastructure (CLI) bytecodes, and it uses a single intermediate representation and global optimization framework for both Java and CLI. The StarJIT compiler is designed to generate optimized code for the major Intel architectures and currently targets two Intel architectures: IA-32 and the Itanium® Processor Family.

In this paper, we describe the overall architecture (bytecode translators, global optimizer, and code generators) of the StarJIT compiler and the design of its intermediate representation, global optimizer, Itanium Processor Family code generator, and dynamic optimization framework. We present implementation details on the single static assignment (SSA)-based global

optimizations [1], the Itanium Processor Family trace scheduler, and the profile-driven dynamic optimization framework.

## INTRODUCTION

Programs targeted to managed runtime environments (MRTEs), such as the Java Virtual Machine and the Common Language Runtime, are distributed in a machine-neutral bytecode format and need to be compiled to native machine code by a dynamic compiler. The performance of managed applications depends on the quality of optimizations and code generation performed by the dynamic compiler. Dynamic compilers, or Just-in-Time (JIT) compilers, are thus a key component of MRTEs.

Because final native code generation happens as part of an application's execution, MRTEs pose several challenges to the dynamic compiler:

1. The dynamic compiler must be sensitive to the time and space efficiency of its optimization algorithms – compilation overheads become overheads on the application's execution. For example, a slow compiler can slow down an application's load time, making the system feel less responsive to the user. A dynamic compiler, therefore, must be designed to balance compilation overhead with code quality.
2. Bugs in the dynamic compiler can become security holes that can be exploited by hackers. MRTEs partially rely on the dynamic compiler to enforce security; for example, the dynamic compiler enforces

---

\* Other brands and names are the property of their respective owners.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

memory safety by inserting checks for type casts and out-of-bound array accesses. Bugs in the dynamic compiler can compromise the safety guarantees provided by the MRTE. A dynamic compiler, therefore, must not only be efficient but also robust.

These challenges are particularly difficult for architectures that rely on compiler optimizations for performance. For example, the Itanium<sup>®</sup> Processor Family architecture relies heavily on expensive and sophisticated code-generation optimizations (such as global scheduling and control speculation) for performance. A dynamic compiler must implement these optimizations robustly and efficiently, and also be flexible, to allow balancing of compilation overhead and code quality.

In comparison to traditional, statically compiled programs, however, MRTEs also provide new performance optimization opportunities:

1. Because native code generation occurs during an application's execution, MRTEs are an ideal environment for dynamic profile-guided optimization. This is important for the Itanium Processor Family, which relies on profile-guided optimizations (such as inlining and trace scheduling) for performance. Dynamic profile-guided optimization also enables the dynamic compiler to concentrate expensive optimizations only on those regions of the program that have the biggest payoffs, thus limiting optimization overhead.
2. The dynamic compiler can tailor the generated code to the platform on which the application is executing. The dynamic compiler can detect platform parameters (such as microarchitecture generation, cache size, and memory size) and tailor the code to the platform parameters. Thus it can deal effectively with "legacy binary" issues.
3. MRTEs provide metadata (such as type information) that can be used for optimization. Metadata gives the compiler precise information about control flow and types used by a program, which the compiler can exploit for optimization (e.g., type-based alias analysis).

We have built the StarJIT compiler as a research infrastructure to investigate these challenges and opportunities on Intel architectures.

The rest of this paper is organized as follows. In the next section, we describe the overall architecture of the StarJIT compiler. We then describe the design of the global

optimizer, including the single static assignment (SSA)-based intermediate representation, global optimization phase structure, and SSA-based global optimization algorithms. We then describe the design of the Itanium Processor Family code generator, including the code generation phase structure and trace scheduler.

## THE ARCHITECTURE OF THE STARJIT COMPILER

The StarJIT compiler is designed to provide a common strongly typed substrate in which code distributed for various managed runtime environments can be safely optimized and targeted to Intel architectures. A further design goal is to enable dynamic profile-driven optimization and recompilation. These goals are reflected directly in the topological organization of the architecture, illustrated in Figure 1. Paths exist connecting every language front-end with every architecture-specific back-end, propagating type information from the source bytecodes through to the architecture-specific back-ends. Furthermore, an additional path for annotating the intermediate representation (IR) used by the global optimizer with profile information from execution of the generated native code enables the seamless injection and use of dynamic information for recompilation.

If virtual machine (VM) support exists, supporting a new hardware architecture for all of the supported languages requires only that a single StarJIT compiler back-end is implemented for that hardware. Similarly, supporting a new language across the supported Intel architectures requires only that a new language front-end be implemented. The primary architectural features of the StarJIT compiler that enable this are divided into language- and architecture-specific portions and language- and architecture-independent portions, both of which are described in this section.

The process for StarJIT compilation follows a single path in this architectural schema, determined by the source language and target architecture. The managed runtime environment (MRTE) bytecode is translated into the global optimizer's IR by the individual front-ends for each source language supported. The language- and architecture-independent portion comprises the global optimizer and the profile feedback manager. The global optimizer is built on an IR called STIR (StarJIT IR). After optimization, architecture-specific code generators translate STIR into architecture-specific IRs, perform architecture-specific scheduling and register allocation, and finally emit the generated native code. A dynamic feedback loop is created through the use of profile information by the Profile Feedback Manager to selectively recompile and guide global and architecture-specific optimization decisions.

---

<sup>®</sup> Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

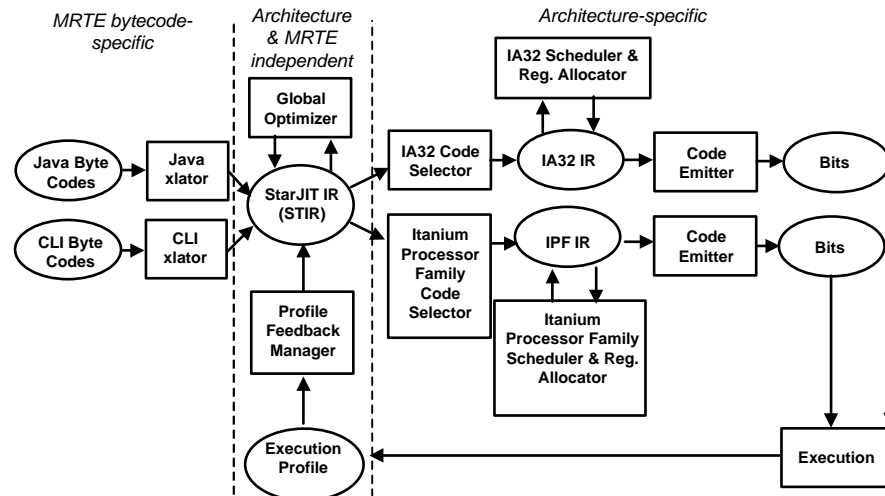


Figure 1: StarJIT compiler architecture

The interface of the StarJIT compiler to a specific MRTE's VM is implemented in a layer that abstracts out the required set of interactions between the JIT and the VM in any MRTE. These include, among other transactions, enumeration of live pointer information for garbage collection, allocation of objects, and metadata queries.

### Java\* and Common Language Infrastructure Bytecode Translators

The initial compilation step is the translation of portable bytecode into STIR. Currently, the StarJIT compiler has bytecode translator front-ends for Common Language Interface (CLI) and Java\*.

Bytecode translation has two phases: the first phase establishes basic block boundaries and exception handling regions, and it recovers type information for variables and operators. There are two major differences in the type information contained in the CLI and Java bytecodes. First, CLI variables are annotated with exact type information whereas Java variables do not have a fixed type and may be reused with different types at different points in the program. Second, CLI operators are untyped whereas Java operators are typed. The first phase of translation reconciles these differences to generate type information for both variables and operators: the Java translator performs type propagation to recover type information for variables, and the CLI translator performs type propagation to recover type information for the operators.

The second translation phase generates STIR and performs simple optimizations, including inlining, constant and copy propagation, folding, strength reduction, type check elimination, devirtualization, elimination of class initialization checks, and value numbering-based redundancy elimination across extended basic blocks.

The bytecode translators generate low-level operators to expose as many calculations as possible to the later global optimization phase. For example, a load of an object field is broken up into component operations that perform a null check of the object reference, load the base address of the object, compute the address of the field, and load the value at that computed address. The front-end translators, however, can be configured to use higher-level operators, which minimizes the need for later-stage coalescing, to take advantage of IA-32's rich addressing modes.

### STIR: The StarJIT Compiler's Intermediate Representation

The StarJIT compiler's intermediate representation (IR) (STIR) is a traditional two-level IR, with control-flow represented as a graph and instructions represented as triples [16].

At a high level, STIR is a control flow graph consisting of nodes and edges. The StarJIT compiler also maintains dominator and loop structure information on this level of IR for use in optimization and code generation. STIR represents both conventional control flow due to jumps and branches, and exceptional control flow due to thrown and caught exceptions, so that the global optimizer and code generators both account for and optimize exceptions and exception handlers. STIR models conventional control flow via basic block nodes and edges, which represent jumps and conditional branches between basic

\* Other brands and names are the property of their respective owners.

block nodes. STIR models exceptional control-flow via dispatch nodes: a thrown exception is represented by an edge from a basic block node to a dispatch node, and a caught exception is represented by an edge from a dispatch node to a block node.

In managed runtime implementations, compiler-generated code generally does not implement exceptional control flow. Instead, the underlying system implicitly handles the exception throws and catches. The StarJIT compiler generates a system call instruction for each throw and registers a handler for each catch. By modeling exceptional control flow explicitly in the control flow graph, the compiler can optimize across throw-catch boundaries. For locally handled exceptions, the compiler replaces expensive throw and catch combinations with cheaper direct branches.

At a lower level, each basic block node consists of a list of instructions, where each instruction is a tuple consisting of an operator and a set of static single assignment (SSA) operands [10]. The operators are low level in order to expose finer-grain operations to the optimizer. SSA form provides explicit use-def links between operands and their defining instructions, which simplifies and speeds up global optimizations. STIR is designed to address both exclusive and dissonant implementation semantics of Java and CLI.

Each STIR instruction and operand is annotated with detailed type information. STIR instructions retain all type information explicit or implicit in the original Java and CLI bytecodes. Optimization passes preserve and update this type information, and they propagate it through to the architecture-specific back-ends for their use. Type information is needed in the code generator to support exact garbage collection (GC), which requires enumeration of the root set at GC safe points. Type information also greatly improves the quality of the compiler analyses by enabling type-based memory disambiguation at various optimization and code-generation stages.

### **Itanium<sup>®</sup> Processor Family and IA-32 Code Generators**

The StarJIT compiler currently supports both the Itanium<sup>®</sup> Processor Family and IA-32 family architectures through distinct back-end code generators. The compiler enables adaptation of new code generators, such as for the Intel<sup>®</sup>

XScale<sup>™</sup> family, through a software interface that allows the optimizer to transparently perform the necessary callbacks to the code generator to construct each code-generator's IR with the appropriate type information.

The propagation of STIR type information and access to metadata provide the code generators with the critical ability to disambiguate memory accesses relatively inexpensively, avoiding aliasing conflicts that would otherwise defeat many code optimizations and transformations. Metadata also allow the code generators to generate sufficient GC information so that the StarJIT compiler can enumerate the root set of live pointers when requested to do so by the garbage collector at runtime.

The implementations of the code generators are completely independent because each architecture family requires a different set of optimizations and code-generation passes and utilizes very different IRs. For example, the Itanium Processor Family code generator performs aggressive trace scheduling; the IA-32 code generator does not need to do this because of its instruction set architecture and its microarchitectural implementation.

### **Dynamic Profile-Guided Optimizations**

The StarJIT compiler supports dynamic profile-guided optimization (DPGO) as part of its dynamic compilation framework. Modern static compilers have used profile-guided optimization (PGO) to achieve significant performance improvement [5] [7]. The performance benefit from PGO on the Itanium Processor Family architecture is even more profound, with a speedup of approximately 20% observed on certain integer benchmarks. Traditional static PGO requires an initial compilation and execution run to collect an execution profile for use in a final compilation step. The three-step process – compiling with instrumentation, executing with representative inputs, and re-compiling with PGO – requires manual involvement, and it adds a significant burden to the usually time-constrained software development cycle. Moreover, this process requires the software vendor to develop a training workload that represents the end-user's workload.

In contrast, DPGO is automatic and transparent to the end user and software vendor. At the center of the StarJIT compiler's DPGO framework is a module called the Profile Manager, which resides in the virtual machine. The Profile Manager manages the collection and processing of the execution profile, and it selects hot methods for recompilation. The first time it compiles a method, the compiler uses lightweight, fast-path optimizations and prepares additional information to support profiling (which depends on the profiling mechanisms used). When a method is executed, its

---

<sup>®</sup> Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

<sup>®</sup> Intel XScale is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

execution profile is collected online. Periodically, the Profile Manager examines the execution profile of each method to determine which methods are hot enough to warrant recompilation. Once the Profile Manager selects a method for recompilation, it tells the compiler to recompile the method with a higher level of optimizations. The Profile Manager also preprocesses the execution profile of the method and provides it to the compiler so that the compiler can apply PGO during recompilation.

This recompilation yields higher performing code for hot methods. The Profile Manager continues this profiling-recompilation process throughout the execution of the application. Since DPGO collects execution profiles and triggers recompilation on-the-fly, it can re-optimize hot methods when there is significant change in their execution profile, allowing the MRTE to adapt to different execution or usage patterns of an application.

Because of the high overhead in collecting an execution profile, DPGO in today's MRTEs is typically constrained to collect a method invocation profile for identifying hot methods and a dynamic call graph for making inlining decisions. As an advanced research platform, the StarJIT compiler provides a much more extensive set of profiles in its DPGO framework. Two types of profiling mechanisms are supported in the StarJIT DPGO framework: one is instrumentation-based and the other is sampling-based.

Instrumentation-based profiling inserts profile-collecting code in the dynamically generated native binary when the compiler first compiles a method. The inserted code increments counters when execution goes through the control flow of the method [2]. The StarJIT compiler currently supports the collection of method invocation and control flow edge execution counts. The inserted code maintains these counters in buffers that are accessible to the Profile Manager. The instrumentation code incurs significant overhead during execution; therefore, the compiler does not generate instrumentation when it recompiles a method with DPGO.

Sampling-based profiling collects an execution profile by collecting samples during the program execution. Instead of simply taking an instruction pointer (IP) sample, the StarJIT compiler utilizes the Performance Monitoring Unit (PMU) of a microprocessor to get a better execution profile of an application. On the Itanium Processor Family architecture, the PMU can monitor and provide a rich set of events and execution information. For example, the Itanium Processor Family PMU has a Branch Trace Buffer (BTrB) that can capture information on the last few branches executed. The branch trace information includes the IP address of a branch instruction and the target IP address of the branch. The information in the BTrB thus captures a short trace fragment during the execution of the program. By taking enough BTrB

samples, the Profile Manager is able to construct an execution profile that approximates the edge profile.

The PMU samples contain virtual IP addresses. To map the sampling profile into a control flow profile, the compiler must map IP addresses into IR at the feedback point. To facilitate such IP-to-IR mapping, the StarJIT compiler emits a basic block mapping table when it dynamically compiles a method. The table allows the mapping of an IP address to a basic block and the branch-target pair of IPs to a control flow edge in the optimizer IR.

The Profile Manager can adjust the overhead of sampling-based profiling by changing the sampling rate. Hence once the compiler has recompiled the majority of hot methods with DPGO, the Profile Manager can tune down the sampling rate to lower the profiling cost. The dynamic adjustment of the sampling rate allows non-stop, low-overhead monitoring of the application, making continuous profiling and recompilation feasible in MRTEs.

When profile information is available, the StarJIT compiler feeds the profile information into the IR, and it selects an optimization path consisting of aggressive profile-guided optimizations. The optimizer propagates the profile information to the Itanium Processor Family code generator so that the code generator can use the profile to guide basic block layout, trace selection, instruction scheduling, and other transformations. We discuss the details of profile usage in later sections.

## GLOBAL OPTIMIZER

The StarJIT compiler uses a single optimization framework for Java<sup>\*</sup> and Common Language Infrastructure (CLI) programs. The StarJIT global optimizer applies a set of classical, object-oriented, and profile-guided optimizations to the method representation, balancing the aggressiveness of optimizations with their compile-time cost.

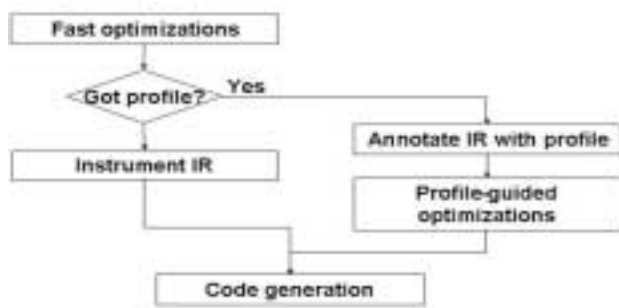
Figure 2 shows the high-level flow of the StarJIT global optimizer. The optimizer has two primary phases. The first phase consists of fast optimizations performed every time the StarJIT compiler is invoked. This phase improves code quality and performance without substantial compile-time cost. It carries out a baseline set of optimizations on all generated code. It is deterministic: it uses no contextual information (such as profiling) that may change in a later recompile. If no profile information is available (i.e., this is the first time the StarJIT compiler

---

<sup>\*</sup> Other brands and names are the property of their respective owners.

is compiling a method) and the Profile Manager is using instrumentation-based profiling, then the StarJIT optimizer instruments the intermediate representation (IR) before invoking the code generator.

If profile information is available (i.e., the method is a hot method that the Profile Manager has selected for recompilation), the optimizer annotates it into the STIR after the first phase and runs the second optimization phase. This phase applies more aggressive optimizations and takes advantage of profile information in the annotated STIR. Through this second phase, the StarJIT compiler focuses compilation time on methods and regions that are most critical to overall performance.



**Figure 2: The StarJIT global optimizer**

Each of the two optimization phases performs the same basic set of optimization passes. These passes are grouped into four categories. *Scope Enhancement* passes, *Privatization* passes, and *Redundancy Elimination* passes are performed in sequence, while *IR Simplification* passes are performed at multiple points to clean up the method representation between passes. In the first phase, all passes use conservative settings to run quickly. In the second phase, the passes use profile information and more aggressive settings. In this manner, the StarJIT compiler balances compile time with performance by concentrating expensive optimizations only on methods that are hot. The remainder of this section describes the optimization passes in more detail.

### Intermediate Representation Simplification Passes

IR simplification passes are a set of very fast optimization passes that the StarJIT optimizer performs several times on the IR. These optimizations reduce the size and complexity of the IR. In addition to improving the code quality, this reduction improves the efficiency of other, more expensive optimizations. IR simplification consists of three passes.

The first pass involves propagation and folding. This pass performs constant, type, and copy propagation over the entire method following the static single assignment (SSA)-form use-def links. As it does this, it also simplifies and folds expressions such as arithmetic on constants or runtime checks for null references that are proven non-null (e.g., a reference defined by a new allocation). When branch conditions or instructions that can potentially raise an exception are folded, the corresponding edges are also removed from the control flow graph, and any unreachable code as a result of the edge deletion is skipped (effectively performing conditional constant propagation [19]).

The second pass eliminates unreachable and useless code. It does the former by testing reachability via traversal from the control flow graph entry; it does the latter by using a sparse liveness traversal over SSA-form use-def links.

The third pass performs fast global value numbering to eliminate common subexpressions [3]. This pass does an in-order depth-first traversal of the dominator tree (instead of the more expensive iterative dataflow analysis done by traditional common subexpression elimination). At any given program point, SSA-form expressions computed earlier within the same basic block are considered available. In addition, expressions that are available at the end of dominating blocks are also available. Expressions that may be killed (such as loads from memory) or have side-effects (such as calls) are ignored.

Global value numbering is effective in eliminating redundant address computation and check instructions (e.g., `chkzero`, `chknull`, and `chkcast` that are redundant or guarded by explicit conditional branches). Later optimization passes eliminate redundant memory accesses (which require alias analysis and kill information) and array bounds checks (which are difficult to remove in a single forward pass because they require arithmetic reasoning and propagation of dataflow facts across loop back edges).

Together, the IR simplification passes can be thought of as a single cleanup pass. This cleanup is performed at a number of points in the optimization process.

### Scope Enhancement Passes

The global optimizer begins with a set of transformations designed to enhance the scope of later optimizations. The first scope enhancement pass normalizes control flow by removing critical edges (a critical edge is an edge from a node with multiple successors to a node with multiple predecessors), and factoring entry and back edges of loops. These transformations prepare the intermediate representation for later optimization; for example, loop

normalization simplifies the implementation of peeling, and critical edge removal is necessary for redundancy elimination.

After normalization, the optimizer performs a set of loop transformations. These include loop inversion, peeling, and unrolling. The first optimization phase is conservative, and performs only loop inversion and limited partial peeling. The second profile-driven optimization phase is more aggressive, and performs profile-driven peeling and unrolling of hot loops. Note that loop peeling, in combination with global value numbering, provides a cheap mechanism to hoist loop-invariant computation and runtime checks.

The third scope enhancement optimization is guarded devirtualization of virtual method calls. Virtual method calls are prevalent in managed runtime environment (MRTE) applications. They differ from direct calls in that the actual call target must be resolved at runtime by examining an object's virtual method table. The costs of this extra level of indirection include the runtime expense of extra code to invoke a virtual method and potentially poorer branch prediction in hardware for that call, as well as the compile-time expense of impeded interprocedural analysis and inlining.

In cases where the optimizer has exact type information, the IR simplification pass is able to devirtualize a virtual call by converting it into a more efficient direct call. In other cases, the target of a virtual method may be highly predictable. In these cases, the scope enhancement pass devirtualizes the call by guarding it with an inexpensive runtime test that checks whether the predicted method is in fact the target. If performed accurately, guarded devirtualization alleviates the runtime costs associated with virtual method calls and enables the compiler to inline targets of virtual method calls. The first phase performs guarded devirtualization conservatively using simple static heuristics. The profile-driven phase performs guarded devirtualization aggressively using block execution and call graph profiles.

The centerpiece of the scope enhancement passes is the inliner. Inlining removes the overhead of a direct call and specializes the called method within the context of its call site. The inliner consists of an iterative process built around the other scope enhancement and IR simplification passes. In the first pass through this cycle, scope enhancement and IR simplification transformations are performed on the original intermediate representation. At this point, the inliner examines each direct call site in the IR (including those exposed by guarded devirtualization), heuristically assigns a benefit to it, and, if it exceeds a certain threshold, registers it in a priority queue. The top candidate, if any, is then selected for inlining. The translator generates IR for the inlined method, and the

cycle is repeated upon the new IR. The inliner then processes the new IR for further inlining candidates (updating the priority queue), splices it into the existing IR, selects a new candidate, and repeats the cycle. The inliner halts once the queue is empty or after the IR reaches a certain size limit. When inlining is completed, the global optimizer performs a final IR simplification pass over the entire intermediate representation.

### Privatization Passes

The privatization passes optimize accesses to memory locations. The privatization phase first performs alias and escape analyses on memory accesses, and then performs synchronization removal [18] and scalar replacement [16]. Alias analysis yields information about which load/store addresses may affect each other [16]. The StarJIT optimizer uses the type information about object fields for alias analysis. For example, accesses to two object fields cannot refer to the same location if the object types, field names, or field types differ. A store cannot alias with a final or read-only field of an object (except in the object constructor). The StarJIT optimizer also uses the definition point of an object reference for alias analysis: a reference to an object that is a method parameter may not alias with a reference resulting from an object allocation.

Escape analysis determines the extent to which accessed memory locations are visible outside the current method [6][13]. Escape analysis determines this information with a sparse SSA-based analysis of each object referenced in a method. An object that is allocated in the body of the method is initially assumed to be private to the method (i.e., non-escaping). Any object passed in as an argument or a return value, passed as an argument to another method, returned as a result, or stored into a static field escapes by definition. Moreover, any object stored as a field in an escaping object transitively escapes. Finally, any object that potentially aliases an escaping object (because of a copy or a merge at an SSA phi node) also escapes. The StarJIT optimizer's current escape analysis algorithm is intra-procedural and relies on the prior inlining pass to expose privatization opportunities. We plan on augmenting the escape analysis pass with inter-procedural information.

Once escape analysis is done, synchronization removal eliminates synchronization operations (which are explicit in STIR) on objects that do not escape a method or that escape only via a return. Scalar replacement promotes object fields and array elements to SSA variables that are amenable to further optimization passes [9]. This pass takes advantage of the alias and escape analysis information to disambiguate memory references.



## Redundancy Elimination Passes

The final set of optimization passes comprises optimizations to eliminate redundant and partially redundant computations. These passes include loop-invariant code motion, bounds-check elimination, and strength reduction [16]. They are deferred until the largest possible program scope is available and the most memory locations have been promoted to scalar variables.

The StarJIT optimizer uses a demand-driven array bounds-check elimination analysis based upon the previously published ABCD algorithm [4]. It first inserts Pi nodes into the IR to split variable live ranges based on branch conditions. Pi nodes capture information gleaned about a variable based on branch conditions. From each variable's definition, the analysis then derives inequality constraints upon that variable's value, which can be used to prove redundancy of bounds checks involving that variable. Unlike the original ABCD algorithm, the StarJIT optimizer's bounds-check elimination implementation does not construct a separate constraint graph, but uses the SSA graph directly to derive constraints during an attempted proof. We also have added handling of symbolic constants to allow check elimination in slightly harder cases, commonly encountered in practice.

To facilitate load hoisting in the code generator, the check-elimination transformations track conditions used to prove that a check can be eliminated. The code generation interface passes this information to the code generator. The scheduler uses this information to determine which branches guard the safety of a given load, and marks the load as speculative if it hoists the load above a guarding branch.

The StarJIT optimizer performs strength reduction to transform expensive operations, such as multiplication by an induction variable in a loop, into simpler operations such as addition. The implementation is based upon the operator strength reduction optimization described in [8], extended to also reduce the strength of memory address computations. The strength reduction pass performs linear function test replacement to eliminate uses of the original loop induction variable in tests (e.g., loop exit tests). This optimization is effective in transforming an iteration through the elements of an array into a series of pointer increments and pointer comparisons, and eliminating the original array index. In cases where the loop index is live after the loop, this pass rematerializes the index on loop exits to still allow removal of the loop index computation from the loop. For the Itanium®

Processor Family architecture, the strength reduction pass can also transform loops with invariant trip counts into counted loops.

The optimizer must be careful during strength reduction because of overflow issues: the optimizer cannot transform a 32-bit integer induction variable used as an array index into a 64-bit pointer (strength reducing the indexing operations) unless it can prove that additions to the 32-bit index will not overflow (and wrap around to a negative number, as required by Java\* bytecode semantics) because adding to the 64-bit pointer will not overflow in the same cases. While unlikely to occur in real code, the induction variable range is checked for possible overflow before such a strength reduction transformation. The range analysis makes use of the same demand-driven bounds-check analysis used for array bounds-check elimination.

## THE ITANIUM PROCESSOR FAMILY CODE GENERATOR

The Itanium® Processor Family code generator is responsible for generating native code for a program represented by STIR. It lowers the program representation to the machine level, performs architecture-dependent optimizations such as register allocation and scheduling, computes the information necessary to support garbage collection (GC), and emits the bits that are directly executed by the processor.

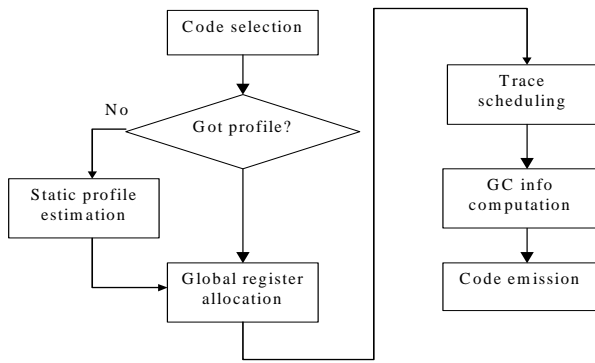
Figure 3 shows the structure of the code generator. The first code generation phase is code selection. During this phase the code generator lowers STIR operations into Itanium Processor Family code sequences and performs simple optimizations such as immediate operand folding, operator folding, and strength reduction. It uses predication [15] to avoid generating additional control flow for complex STIR operations such as `instanceOf`. The Itanium Processor Family instruction sequences generated from STIR usually contain many operations that move data between temporaries, variables, incoming and outgoing arguments, and return values. The code selector makes a pass over the intermediate representation to coalesce the sources and destination operands of moves, and to remove the resulting redundant moves.

---

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

---

\* Other brands and names are the property of their respective owners.



**Figure 3: Itanium Processor Family code generator phases**

The optimizer drives code selection through a code-generation interface. This interface abstracts the information that the optimizer should communicate to a code generator from the details of STIR implementation and allows the code generator to be used with any front-end that supports the code generation interface. The subsequent code generation phases require profile information to guide optimizations. When dynamic profile information is not available, the code generator estimates the profile using static heuristics [1].

The ordering of register allocation and code scheduling is a classical phase-ordering problem [12]. Register allocation performed before scheduling introduces additional anti and output dependencies that restrict scheduler freedom to reorder the instructions. Register allocation performed after code scheduling may require an additional scheduling pass to accommodate generated spill code. In addition, register allocation quality may suffer because of increased register pressure. The code generator chooses a middle-ground approach. It divides all operands into two categories: local and global. An operand is local if it has a single definition and its live range does not span a loop boundary. All other operands are global. Only global operands require iterative data flow analysis to compute their liveness. The liveness of local operands can be computed with a single reverse pass over the IR. The global operands are assigned registers during the global register allocation phase that occurs before scheduling. This introduces only a few data dependencies, as most of the operands are local. The local register allocator is integrated with scheduling. The scheduler keeps track of the register pressure, and materializes and schedules spill code as needed.

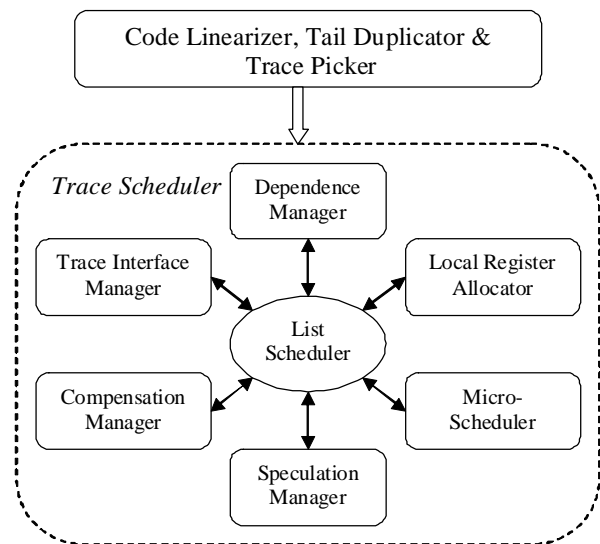
The code scheduler is the most complex component of the code generator. In addition to scheduling instructions using trace scheduling [11], it performs code layout and local register allocation. The design of the trace scheduler is described in the next section.

After scheduling, the code generator computes the information necessary to support GC. For each call instruction, it computes the set of registers and stack locations that contain live references and interior pointers (pointers to the middle of the objects allocated on the heap), and it records this information in a data structure called the GC map table. During garbage collection, the garbage collector enumerates the root set by iterating over the set of frames on each thread's runtime stack. For each frame, the garbage collector makes a callback into the JIT compiler asking it to enumerate the set of live references for that frame and to unwind to the previous frame. The JIT compiler computes the set of live references for the frame using the GC map information.

The final code emission phase emits the native Itanium Processor Family binary code into memory for execution. This phase also emits the GC map table, exception handler tables (for dispatching exceptions), stack unwinding information (for root set enumeration, exception unwinding, and runtime security checks), and the IP-to-IR mapping tables (for profile gathering).

### Trace Scheduler Design

The modular trace scheduler design facilitates managed runtime environment (MRTE) research work, retargetability to other micro-architectures, and portability for use in other virtual machine (VM) or compilation systems. The various components of the trace scheduler are shown in Figure 4. The components have been designed as independent modules with clear interfaces so that they can be applied to each trace selectively.



**Figure 4: Trace scheduler components**

The first pass of the trace scheduler is the code linearizer, tail duplicator, and trace picker. Trace selection is important because it defines the scheduling scope. Trace selection and scheduling are best done after code (basic block) layout, to schedule unconditional branches introduced by code linearization, and to form cross-block bundles and cycles. Tail duplication is useful to eliminate side entries into traces, but must be done before code layout decisions are finalized. Tail duplication decisions, however, are best made with input from trace formation. Therefore, there is a cyclic phase ordering dependency between trace picking, code layout, and tail duplication.

The StarJIT Itanium Processor Family code generator uses a novel scheme that performs all three together. The code layout technique is a top-down scheme similar to that described by Pettis & Hansen [17]. Code layout, trace formation, and tail duplication decisions all benefit from any available branch profile information. Code layout uses profiles to improve cache locality and reduce taken branches. Along hot paths the trace picker picks longer traces, and the tail duplicator is more aggressive in removing cold side entries, while on cooler paths shorter traces are picked with little or no tail duplication. Finally, a few compensation blocks are added on some critical edges. After scheduling, the code generator eliminates useless compensation blocks, which have no compensation code moved into them.

At the core of the trace scheduler is the list scheduler, which schedules one trace at a time. The list scheduler schedules instructions from a data-ready list. It uses several heuristics to choose between data-ready candidates. These include critical path length, slack (a measure of the freedom to delay an operation without delaying the overall schedule), register and resource availability and future needs, code size and code motion usefulness metrics, and effects of any required compensation, or speculation. The heuristics are profile sensitive: their basic goals are to generate high-performance code at hot traces and to enable fast generation of compact code at cold traces. The list scheduler heuristics also guide multiway branch generation. MRTE safety checks, such as null pointer, array bounds, and type checks, result in a large number of branch operations. It is therefore important to bundle multiple branches together to reduce code size, control height, and mispredicted branches.

The list scheduler uses a micro-scheduler to schedule instructions within a cycle. The micro-scheduler models resources and dispersal rules, and makes compact bundling decisions. For the Itanium Processor Family, it is important to integrate scheduling with bundling because the bundling choices influence dispersal. The micro-scheduler is based on the Open Research Compiler's

micro-scheduler [14]. It abstracts away the machine details and reads the Itanium micro-architecture definition from a knobs file.

The dependence manager tracks all register data dependencies, memory dependencies, and control dependencies while trying to avoid transitive dependencies, for efficiency reasons. It uses MRTE metadata to avoid creating false memory and control dependencies. Memory disambiguation is based on the properties of pointers to memory locations such as type, memory region (heap, stack, static), and access semantics (e.g., field, array element). The dependence manager uses the safety semantics of MRTE memory operations to avoid unnecessary control dependencies. A load is safe (i.e., can be issued without making it speculative) everywhere except before its corresponding safety checks (chknul, chkbound, and/or chkcast). When the optimizer combines or eliminates any of these checks based on control or data flow implications, it keeps around enough information to allow the dependence manager to recognize control dependencies of such loads on the appropriate check and/or branch instructions. The dependence manager also enables the list scheduler to use predication to convert a control dependency on a branch to a data dependency on the associated predicate-generating compare. This allows the list scheduler to predicate a block partially, thus reducing the need for speculation, check, and recovery generation.

The speculation manager uses the Itanium Processor Family control speculation feature to schedule loads before the branches on which they are control dependent. It keeps track of the speculative loads and dependent speculative instructions that should be included in the recovery code. After all traces have been scheduled, the speculation manager materializes the recovery code and schedules it using a local scheduler.

When instructions are moved above a trace side entry or below a trace side exit, the compensation manager inserts copies of these instructions in the off-trace blocks. The scheduler performs code motion, only when heuristics suggest that the good done to the on-trace path is not outweighed by any harm done by compensation code to the off-trace path. Code motion requiring compensation insertion into previously scheduled traces is not permitted. Profile information (which determines the order in which traces are scheduled), therefore, guides compensation code decisions. The compensation manager also avoids compensation code when control and data dependence relationships indicate that it is unnecessary. For example, compensation code is not needed at intermediate side entry points when an instruction is moved to a dominating point in the trace and the instruction's operands are not modified on any off-trace path.

The trace interface manager models liveness and data flow latency across trace boundaries (trace main entry/exit and side entries/exits), thus maximizing scheduling freedom and improving performance at trace interfaces.

The StarJIT trace scheduler has an integrated local register allocation module (as mentioned earlier, global operands are allocated registers prior to scheduling). This module monitors liveness of local temporaries and allocates registers to them when their definitions are scheduled. A local temporary has a single definition that dominates all its uses. The scheduler exploits this property to model register pressure during scheduling, and to materialize and schedule spill code on-the-fly, thus performing efficient and optimized register allocation.

## CONCLUSION

Managed Runtime Environments (MRTes) depend on dynamic compilation for performance and security. The strict runtime requirements of dynamic compilation pose new challenges to compiler engineers. These requirements also provide new dynamic optimization opportunities involving both the compiler and the hardware.

In this paper, we have described the design of the StarJIT compiler. Built upon a framework that enables dynamic recompilation for a range of MRTes and Intel architectures, this research infrastructure enables heretofore intractable research opportunities in implementation tradeoffs of managed runtimes and hardware architectures.

## ACKNOWLEDGMENTS

The authors thank members of the Open Research Platform (ORP) VM team, Michal Cierniak, Neal Glew, Rick Hudson, Brian Lewis, James Stichnoth, Sreenivas Subramoney, and Weldon Washburn. The StarJIT compiler would not have been realized without their support and efforts in making the ORP VM robust and high-performing. We thank Youfeng Wu, Roy Ju, and Sun Chan for providing valuable feedback on the IPF trace scheduler design and details on the ORC micro-scheduler. The authors also thank Jesse Fang for his guidance and continuing support of this work, Ken Lueh for his early contributions to the StarJIT source code, and Youngsoo Choi for his contributions to the Itanium Processor Family PMU driver.

## REFERENCES

- [1] T. Ball and J.R. Larus, "Branch Prediction for Free," in proceedings of *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 300-313.
- [2] T. Ball and J.R. Larus, "Optimally Profiling and Tracing Programs," *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, January 1992, pp. 59-70.
- [3] P. Briggs, K.D., Cooper and L.T. Simpson, "Value Numbering. Software-Practice and Experience," vol. 27(6), June 1997, pp. 701-724.
- [4] R. Bodik, R. Gupta, and V. Sarkar, "ABCD: Eliminating Array-Bounds Checks on Demand," in proceedings of the *SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000, pp. 321-333.
- [5] P.P. Chang, S.A. Mahlke and W.W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software-Practice and Experience*, vol. 21(12), Dec. 1991, pp.1301-1321.
- [6] J.-D. Choi, M. Gupta, M.J. Serrano, V.C. Sreedhar and S.P. Midkiff, "Escape Analysis for Java," in proceedings of the *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, 1999, pp. 1-19.
- [7] R. Cohn, D. Goodwin and P.G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike," *Digital Technical Journal*, vol. 9, No. 4, 1997, pp. 3-20.
- [8] K.D. Cooper, L.T. Simpson and C.A. Vick, "Operator Strength Reduction," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 5, September 2001, pp. 603-625.
- [9] K.D. Cooper and L. Xu, "An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations," *ACM 2002, Workshop on Memory System Performance (MSP '02)*, Berlin, Germany, June 16, 2002.
- [10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, No. 14, October 1991, pp 451-490.
- [11] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, C-30(7), July 1981, pp. 478-490.
- [12] S.M. Freudenberger and J.C. Ruttenberg, "Phase Ordering of Register Allocation and Instruction Scheduling," in proceedings of the *International Workshop on Code Generation*, May 1991, pp. 146-172.
- [13] D. Gay and B. Steensgaard, "Fast Escape Analysis and Stack Allocation for Object-Based Programs," 9<sup>th</sup>

*International Conference on Compiler Construction*, (CC '2000), Springer-Verlag, Vol. 1781, 2000, pp. 82-93.

- [14] R. Ju, S. Chan, F. Chow, X. Feng and W. Chen, "Open Research Compiler (ORC) Beyond Version 1.0," tutorial presented at *PACT-2002*, September 22, 2002.
- [15] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," in proceedings of the 25<sup>th</sup> *International Symposium on Microarchitecture* (MICRO 25), Dec. 1992, pp. 45-54.
- [16] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, CA, 1997.
- [17] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," in proceedings of the *ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, White Plains, N.Y., June 20-22, 1990, pp. 16-27.
- [18] E. Ruf, "Effective synchronization removal for Java," in proceedings of the *ACM SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, British Columbia, June 2000, pp. 208-218.
- [19] M. Wegmen and F. Zadeck, "Constant Propagation with Conditional Branches," *ACM Transactions on Programming Languages and Systems*, vol. 13, No.2, April 1991, pp. 181-210.

## AUTHORS' BIOGRAPHIES

**Ali-Reza Adl-Tabatabai** is a senior staff researcher in the Programming Systems Lab. He received a B.Sc. degree from UCLA in Computer Science & Engineering and a Ph.D. degree from Carnegie Mellon University in Computer Science. His research interests include dynamic compilation and optimization, managed runtimes, memory hierarchy design, and compression. His e-mail is [ali-reza.adl-tabatabai@intel.com](mailto:ali-reza.adl-tabatabai@intel.com)

**Jay Bharadwaj** is a senior staff researcher in the Programming Systems Lab. He received a B.S. degree in Mechanical Engineering from IIT Madras, India and M.S. degrees in Computer Science and Mechanical Engineering from Rensselaer Polytechnic Institute and SUNY Stony Brook, respectively. His research interests include managed runtimes, hardware software cooperation, and compilation techniques. Other interests include activities

requiring use of hand or power tools. His e-mail is [jay.bharadwaj@intel.com](mailto:jay.bharadwaj@intel.com)

**Dong-Yuan Chen** is a staff researcher with the Programming Systems Lab. He received his Ph.D. degree in Computer Science from Yale University in 1995. He has worked on back-end compiler optimizations, including software pipelining and machine modeling, and various microarchitectural performance studies for the Itanium Processor Family architecture. His current interests include lightweight online profiling mechanisms and dynamic profile-guided optimizations in managed runtime environments. His e-mail is [dong-yuan.chen@intel.com](mailto:dong-yuan.chen@intel.com)

**Anwar Ghuloum** is a senior staff researcher in the Programming Systems Lab. He received a B.Sc. degree from UCLA in Computer Science & Engineering and a Ph.D. degree from Carnegie Mellon University in Computer Science. His research interests include managed runtime environments, memory hierarchy design, and compression. Other pursuits include cycling, tri, building bikes, painting, and the uses of coherent light. His e-mail is [anwar.ghuloum@intel.com](mailto:anwar.ghuloum@intel.com)

**Vijay Menon** is a staff researcher in the Programming Systems Lab. He received a B.S. from the University of California, Berkeley in Electrical Engineering and Computer Science and a Ph.D. from Cornell in Computer Science. His current research interests include program analysis, dynamic compilation, and managed runtime environments. His e-mail is [vijay.menon@intel.com](mailto:vijay.menon@intel.com).

**Brian R. Murphy** is a Just-In-Time researcher at Intel Labs. He has done analysis of functional languages, automatic parallelization of Fortran code, development of advanced program analysis techniques, programming language design and implementation, Unix and Linux systems programming and administration, and Web site development and management. He received S.B. and S.M. degrees from M.I.T., and a Ph.D. degree from Stanford University. His e-mail is [brian.r.murphy@intel.com](mailto:brian.r.murphy@intel.com)

**Mauricio Serrano** received his Ph.D. degree in Computer Engineering from the University of California Santa Barbara in 1994, an M.S. degree from Rensselaer Polytechnic Institute, and a B.S.E.E. from Javeriana University, Bogota, Colombia. Before joining Intel, he spent several years working with IBM T.J. Watson/New York and STL/San Jose, where he worked in several compiler areas including program restructuring, retargetable code generation, and Java performance optimizations. His other interests are computer architecture and performance modeling. He published the first dissertation on SMT (Simultaneous Multithreaded Processors) in 1994, although at that time he called it

SMS (Simultaneous Multistream Superscalar Processors).  
His e-mail is [mauricio.j.serrano@intel.com](mailto:mauricio.j.serrano@intel.com)

**Tatiana Shpeisman** is a staff researcher in the Programming Systems Lab. She received her B.Sc. degree from the Leningrad Electrical Engineering Institute in Applied Mathematics and M.S. and Ph.D. degrees from the University of Maryland, College Park, in Computer Science. Her research interests include compilation techniques, managed runtimes, and sparse matrix computations. Her other interests include hiking in the Sierras, ballroom dancing, and classical ballet. Her e-mail is [tatiana.shpeisman@intel.com](mailto:tatiana.shpeisman@intel.com).

Copyright © Intel Corporation 2003. This publication  
was downloaded from <http://developer.intel.com/>.

Legal notices at  
<http://www.intel.com/sites/corporate/tradmarx.htm>.

For further information visit:

[developer.intel.com/technology/itj/index.htm](http://developer.intel.com/technology/itj/index.htm)