

METHOD AND APPARATUS FOR GENERATING OPTIMIZED CODE WITH COMPOUND INSTRUCTION BY ELIMINATING NULLIFIED COMPOUND INSTRUCTION

CROSS-REFERENCE TO RELATED PATENT APPLICATION

This application claims the benefit of Korean Patent Application No. 10-2009-0070997, filed on July 31, 2009, in the Korean Intellectual Property Office, the disclosure of which is incorporated herein in its entirety by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a method and apparatus for generating an optimized code with a compound instruction by eliminating nullified compound instruction during compound instruction selection, and more particularly, an apparatus for generating code that iterates compound instruction selection of a instruction selector and register allocation (including register coalescing) of a register allocator until a nullified compound instruction (NCI) does not exist any more, and a method of generating code of the apparatus.

2. Description of the Related Art

Generally, a processor loads data stored in a memory to a register, for program execution. In order to execute a program, program parameters (symbols) need to be allocated to the register installed in the processor. These operations are performed by a compiler for converting a source code into a target code. A code-generation operation of the compiler mostly includes an instruction selection operation and a register allocation operation. In the register allocation operation, the compiler uses a graph coloring algorithm.

Most embedded processors such as a digital signal processor (DSP) use a small number of registers dedicated to a function block, for good performance. In this architecture, values that are stored as an output of one function block in a register are frequently moved to another register so as to be input to another function block. Thus, also in the code-generation operation, many copy-related instructions

(copy or move) are used. Since the copy-related instructions reduce the performance of codes, optimization technologies such as register coalescing with the graph coloring algorithm are used, thereby improving the performance of codes generated by the compiler.

5 A compound instruction has been regarded as an efficient way of improving performance, and encodes several arithmetic and logic units (ALUs) or memory operations within an instruction word. Cores embedded in a system-on-a-chip (SOC) suffer from severe design constraints on performance, energy and storage. At the same time, the cores are also required to be programmable in order to ensure
10 reusability towards different applications. To meet all these design challenges, they are often implemented in the form of application-specific instruction-set processors (ASIPs).

The instructions of ASIP are normally compound instructions such as add-mult and shift-move, each of which encodes more than one ALU or memory
15 operations. Their main advantage for embedded processor cores is code size reduction in that several operations are packed inside one instruction word. In addition, they are usually designed to perform their internal operations in sequence or parallel within one machine cycle.

Since the extra sequencing overhead needed for multi-cycle instructions may
20 be avoided, it is more efficient to execute one compound instruction for a set of operations than to separately execute multiple instructions for them. Therefore, if compound instructions are fully utilized, the running time may be reduced.

However, such advantages of compound instructions come with a downside in that they complicate the code generation of the compiler. Although the register
25 coalescing is a useful optimization algorithm, the register coalescing sometimes creates an unexpected side effect such as nullified compound instruction when the register coalescing is applied to the code with compound instructions.

SUMMARY OF THE INVENTION

30 The present invention provides a method and apparatus for generating an optimized code with a compound instruction, in which the quality of a final target code may be improved by preventing generation of a nullified compound instruction (NCI) by considering register coalescing used at a register allocation operation during compound instruction selection.

According to an aspect of the present invention, there is provided an apparatus for generating an optimized code with a compound instruction by eliminating nullified compound instruction during compound instruction selection, including an instruction selector for selecting a compatible compound instruction candidate from a candidate interference graph consisting of compound instruction candidates extracted from an intermediate representation tree; a register allocator for generating a code to which a register is allocated by allocating and merging a register to be used in the compound instruction candidate selected by the instruction selector; a nullified compound instruction detector for detecting a nullified compound instruction including a nullified move instruction in the code to which the register is allocated; and a final code generator for emitting the code to which the register is allocated as a final code, when the nullified compound instruction is not detected, wherein the nullified compound instruction detector eliminates the selected compound instruction candidate, and allows the instruction selector to select a new compound instruction candidate, when the nullified compound instruction is detected.

According to another aspect of the present invention, there is provided a method of generating code of the apparatus.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and other features and advantages of the present invention will become more apparent by describing in detail exemplary embodiments thereof with reference to the attached drawings in which:

FIG. 1 is a block diagram of a code generating apparatus according to an embodiment of the present invention;

FIG. 2 is a block diagram of an instruction selector of FIG. 1, according to an embodiment of the present invention;

FIG. 3 is a block diagram of a code generating apparatus according to another embodiment of the present invention;

FIGS. 4A and 4B are diagrams of intermediate representation (IR) trees and compound instruction candidates (CICs), according to an embodiment of the present invention;

FIGS. 5A through 5E are diagrams of a candidate interference graph (CIG), a labeling operation, a covering operation, a code at the covering operation, and a

code generated after a register allocation operation, respectively, according to an embodiment of the present invention;

FIGS. 6A through 6E are diagrams of a CIG, a labeling operation, a covering operation, a code at the covering operation, and a code generated after a register allocation operation, respectively, according to another embodiment of the present invention;

FIGS. 7A through 7E are diagrams of a CIG, a labeling operation, a covering operation, a code at the covering operation, and a code generated after a register allocation operation, respectively, according to another embodiment of the present invention; and

FIG. 8 is a flowchart of a method of generating a code, according to an embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Hereinafter, the present invention will be described in detail by explaining exemplary embodiments of thereof with reference to the attached drawings. Like reference numerals in the drawings denote like elements.

FIG. 1 is a block diagram of a code generating apparatus 100 for optimizing a code by eliminating a nullified compound instruction (NCI) during the compound instruction selection, according to an embodiment of the present invention.

The code generating apparatus 100 includes an instruction selector 110, a register allocator 120, a nullified compound instruction (NCI) detector 130 and a final code generator 140.

The code generating apparatus 100 is a kind of a compiler for receiving a source code as an input and finally generating a target code understandable by a machine, and uses a tree parsing method and an active programming method. The code generating apparatus 100 may improve the quality of the target code by preventing generation of the NCI by considering register coalescing used at a register allocation operation during the compound instruction selection.

The instruction selector 110 selects instructions from a target processor, having the same semantic as that of the source code written by a user.

The register allocator 120 generates a code containing compound instructions by allocating and merging a register used by the selected instructions from the target

processor. The generated code is transmitted to the NCI detector 130 and the final code generator 140.

In a register allocation algorithm, an interference graph IG is used to represent the live range of each operand value as a node (or a temporary). The IG visualizes with an edge the interference of the live range of two temporaries. A node in the IG is marked if it is trivially colorable, meaning that its corresponding temporary may be assigned any register, irrespective of the register assignments of its neighbors. It has a significant degree if the number of its neighbors is higher than that of the available colors/registers; otherwise, it has a low degree. Only the nodes of low degree may be trivially colored.

In the graph coloring, register coalescing merges two nodes respectively representing the source and destination operands of a move into a new node, and adjacent edges of the new node are united with adjacent edges of the nodes being replaced. Then, when a register is assigned to the new node, the related move instruction may be eliminated. As such, register coalescing helps the compiler to eliminate many redundant moves in the code. However, although the register coalescing is a useful optimization algorithm, the register coalescing sometimes creates an unexpected side effect such as nullified compound instruction when the register coalescing is applied to the code with compound instructions.

The NCI detector 130 detects whether the NCI is generated in the code generated by the register allocator 120. If the NCI is not detected, the NCI detector 130 allows the final code generator 140 to generate a final code. If the NCI is detected, the NCI detector 130 notifies the instruction selector 110 about the detection of the NCI and allows the instruction selector 110 to reselect instructions.

The code generating apparatus 100 repeatedly performs the instruction selection operation of the instruction selector 110 and the register allocation operation (including a register coalescing operation) of the register allocator 120 until the NCI is completely eliminated.

FIG. 2 is a block diagram of the instruction selector 110 of FIG. 1, according to an embodiment of the present invention.

Referring to FIG. 2, the instruction selector 110 includes a compound instruction candidate (CIC) set selector 111, a labeling unit 112 and a covering unit 113.

The CIC set selector 111 checks and selects a node cluster that is a compound instruction candidate (CIC) in an intermediate representation (IR) tree.

The labeling unit 112 labels each node with simple rules in the IR tree, and the covering unit 113 generates a code covering the IR tree.

FIG. 3 is a block diagram of a code generating apparatus 300 according to another embodiment of the present invention.

The code generating apparatus 300 may include a code generator 100 and a code generator-generator 200. The code generator 100 have the same configuration and function as those of the code generating apparatus 100 of FIG. 1, and thus details thereof are not repeated.

The code generator-generator 200 includes a split rule extractor 210 and a code generator builder 220.

The code generator-generator 200 splits nontree-shaped rules for compound instructions into sub-rules of tree shape, and automatically builds the code generator 100.

Since the nontree-shaped rules for compound instructions are not directly applicable to a tree parser, the split rule extractor 210 splits all of the nontree-shaped rules for compound instructions into the sub-rules of tree shape. In this case, the sub-rules are defined as split rules, the nontree-shaped rules for compound instructions are defined as complex rules, and tree-shaped rules for either basic or compound instructions are defined as simple rules.

The code generator builder 220 automatically builds the code generator 100 from a target processor description written in a tree grammar.

A rule example is as follows.

(Example)

rules : $r_1(\text{add},1)$, $r_2(\text{mov},1)$, $r_3(\text{sll},1)$, $r_4(\text{sll-add},1.5)$, $r_5(\text{sll-mov},1)$, $r_6(\text{add-move},1)$

split rules : $r_{41}(\text{sll})$, $r_{42}(\text{add})$, $r_{51}(\text{sll})$, $r_{52}(\text{mov})$, $r_{61}(\text{add})$, $r_{62}(\text{mov})$

In Example, a rule r_i is accompanied by a tuple as a set of values that are sequentially arranged. A first element of the tuple represents an operation, and a second element represents associated cost of the operation. For example, a rule r_6 consists of two parallel operations 'add' and 'mov', and its associated cost is one.

In Example, while r_1 , r_2 and r_3 are simple rules, r_4 , r_5 and r_6 are complex rules. Thus, r_4 , r_5 and r_6 are divided respectively into (r_{41}, r_{42}) , (r_{51}, r_{52}) and (r_{61}, r_{62}) . At a split rule extraction (SRE) operation, their costs are not yet determined.

The code generator-generator 200 stores all the rules including split rules into the new built instruction selector 110 of the code generator 100. In this case, the information is collected about which split rules may be correctly grouped to form a complex rule.

FIGS. 4A and 4B are diagrams of IR trees and CICs with respect to Example, according to an embodiment of the present invention. The IR trees and the CICs of FIGS. 4A and 4B use Example.

The CIC set selector 111 identifies a node cluster in the IR trees, called a CIC. For example, in the IR trees of FIG. 4A, a node cluster B including a tree node 1 and a tree node 4 may be a CIC. This is because these member nodes are matched respectively to the rules r_{41} and r_{42} split from the complex rule r_4 .

Likewise, node clusters D, E, F and G may be CICs. On the other hand, node clusters A, C and H are removed from the candidate list since the node clusters A, C and H may not be executed in parallel due to data dependence.

When a covering unit 113 for performing precover and cover operations covers the IR trees, the CICs are used. Then, the final code generator 140 generates the CICs as multi-output instructions (MOIs) that are compound instructions with parallel operations, and emits the MOI as an output code.

However, due to some interference between the CICs, not all CICs are selected for being considered at the covering operation. For example, although five CICs are computed in FIG. 4B, the node clusters B and D are not simultaneously selected. This is because, if the node clusters B and D are simultaneously selected, the tree node 1 covered twice.

For valid covering in such a way that none of the selected CICs interfere each other, the CICs may be carefully selected. At a candidate set selection (CSS) operation of the CIC set selector 111, interference on a candidate interference graph (CIG) may be classified into modeled into two types, and the interference may be properly handled according to their types.

In the CIG, a node 'n' stands for a CIC, and an edge stands for interference between adjacent nodes, enforcing only one of the nodes to be selected.

FIGS. 5A through 5E are diagrams for explaining a case where CICs are generated with respect to Example, according to an embodiment of the present invention. FIGS. 5A through 5E are diagrams of a CIG, a labeling operation, a covering operation, a code at the covering operation, and a code generated after a register allocation operation, respectively, with respect to Example.

In FIG. 5A as the CIG, edges denoted by sold lines represents covering interference. Like in a case of B and D of FIG. 5B, since CICs connected by the edges denoted by the sold lines share the same split rule, the two CICs raises a conflict at the covering operation.

Edges denoted by dotted lines in FIG. 5A represents dependence interference. The dependence interference comes to exist between two CICs if one CIC is selected in order to cover IR trees brings new dependence among other nodes.

For example, it is assumed that the CIC D is selected in order to cover the tree nodes 1 and 5 in FIG. 4A. Selecting of the CIC D introduces data dependence between tree nodes 3 and 4 of G passing through the nodes of D. Since this dependence serializes the execution of internal nodes, G may be no longer selected for a generation of a MOI in the code.

In the CIG, each CIC node 'm' is weighted. The weight w_m for the complex instruction corresponding to 'm' is defined by opportunity cost C_0 of a complex rule 'r' of Equation 1 below:

$$w_m = C_0(r) = \sum C_s(r_i) - C_c \dots\dots\dots(1),$$

where C_c is the cost of the compound instruction in terms of size and latency.

When r_i is the i-th split rule from r, it is assumed that r'_i is a simple rule representing a simple rule representing the same instruction pattern as r_i . Then, the two rules r_i and r'_i are equivalent. $C_s(r_i)$ is defined to be the cost of r'_i , and $\sum C_s(r_i)$ is defined to be the total cost of all the simple rules each corresponding to a split rule r. For example, it may be seen that $C_s(r_{41})=1$ in the ruled of Example, which is equal to the cost of an instruction 'sl' of r_3 .

Each cost of r_4 , r_5 and r_6 is as follows.

$$C_0(r_4) = (1+1)-1.5 = 0.5, C_0(r_5) = (1+1)-1 = 1, C_0(r_6) = (1+1)-1 = 1$$

Accordingly, weights of the CICs are calculated as follows.

$$w_B = 0.5 \text{ and } w_D = w_E = w_F = w_G = 1$$

The main task of the CSS operation is to find a set of compatible CICs with maximum profit (or maximum weight sum). If two CICs obey interference constraints, the two CICs are compatible CICs.

In FIG. 5A, four set of compatible CICs are exemplified: {B, E}, {B, F}, {D, E} and {F, G}. The two sets of {D, E} and {F, G} equally have the maximum profit (=2). It is assumed that {F, G} is selected at the CSS operation.

Next, the CIC set is utilized by the tree parser that traverses the IR trees, and dynamic programming is used in order to efficiently generate the instructions for the final code. Although the CICs with maximum profit do not guarantee the tree parser the code with optimal cost, the CICs alt least provide the tree parser with a good starting point for searching for the optimality. However, the complex rule is matching to a CIC is nontree shaped. Thus, the rule as a whole may not be recognized by the tree parser. The only way for the rule to participate in the parsing is via the tree-shaped split rules. For this reason, after an optimal CIC set is computed, the cost of every split rule r_i from the complex rule matching to each CIC in the set may be calculated by Equation 2 below:

$$C(r_i) = C(r'_i) - C_0(r)/n \dots\dots\dots(2),$$

Where r_i is a i -th split rule, r'_i is a simple rule equivalent to r_i , and n is the number of split rules 'r'.

Since F and G in the selected CIC set are matching to the complex rule r_6 , the costs of the split rules r_{61} and r_{62} are calculated from the costs of r_6 , as follows:

$$C(r_{61}) = 1 - 1/2 = 0.5, C(r_{62}) = 1 - 1/2 = 0.5$$

rules : $r_1(\text{add}, 1)$, $r_2(\text{mov}, 1)$, $r_3(\text{sll}, 1)$, $r_{61}(\text{add}, 0.5)$, $r_{62}(\text{mov}, 0.5)$

After the costs of the split rules are calculated, the simple rules are labeled to each node in the IR trees at the labeling operation.

At the covering operation subsequent to the labeling operation, the tree parser picks an optimal derivation covering the whole IR tree, among the rules labeled together at each node. For example, the node 3 in FIG. 5B is labeled with two rules r_2 and r_{62} .

FIG. 5C shows an IR tree consisting of three rules covering every node: $r_3 < -(1)$, $r_6 < -(2, 5)$, and $r_6 < -(3, 4)$. As a result, three instructions 'sll', 'addmov' and 'addmov' are obtained.

FIG. 5D shows the code at the covering operation, and FIG. 5E shows the code generated after applying register coalescing and allocating to the code of FIG.

5D. In this case, since an instruction 'move' alone may be separated and erased at the register allocation operation, the compiler may no longer get rid of the useless 'move' contained in the instruction 'addmov' on the first line.

As shown in FIG. 1, the register allocator 120 has a feedback path to the instruction selector 110 in order to inform the instruction selector 110 about the existence of the NCI. This function is performed by the NCI detector 130. The NCI detector 130 detects the NCI, and then allows the instruction selector 110 to eliminate the CICs F and G responsible for the NIC from the CIG and to select new CIC. Before the register is allocated, although the code of FIG. 5D was an optimal code having the cost '3', since the code of FIG. 5D generates the NCI like in FIG. 5E, the code of FIG. 5D may be eliminated.

After the NCI detector 130 detects the NCI, the instruction selector 110 eliminates the CICs responsible for the NIC from the CIG and selects the next most compatible CIC set. However, this method is too aggressive and naive solution to address the NIC problem.

FIGS. 6A through 6E are diagrams for explaining a case where CICs are generated with respect to Example, according to another embodiment of the present invention. FIGS. 6A through 6E are diagrams of a CIG, a labeling operation, a covering operation, a code at the covering operation, and a code generated after a register allocation operation, respectively, with respect to Example.

Even if a CIC 'X' has the NIC in a first trial, there is still a high probability that the CIC is not nullified again as far as X is selected with different CICs and forms a new compatible CIC set for the next trial. Different CICs may produce different results not only of the covering operation but also of the register coalescing operation. Only because the NCI is generated, simply eliminating 'X' may deprive a chance to discover an optimal code by using 'X'.

This case is shown in FIG. 6A. Interference edges may be added to the CIG in order to address the NCI problem.

Once the first choice {F, G} fails due to the NCI 'addmov' in the code of FIG. 5E, the instruction selector 110 selects the next most compatible CIC set with maximum profit.

With regard to the same CIG, there is a change to select {F, G} again since the set is still among CIC sets with maximum profits. In order to prevent {F, G} from

being selected, a new edge is added to the CIG, and the new edge is denoted by a thick line in FIG. 6A, for representing a different type of interference.

The interference edge denoted by the thick line indicates a mutual exclusion edge between adjacent CIC nodes. In order to prevent the generation of the NCI, nodes connected by the mutual exclusion edge may not be selected together as a CIC set, in any circumstances.

Now, four compatible CIC sets {B, E}, {B, F}, {D, E} and {G} exist, and among which {D, E} is the only CIC set with maximum profit 2.

In a second iteration, after the labeling operation of FIG. 6B and the covering operation of FIG. 6C are performed, the code of FIG. 6D is generated. FIG. 6D shows the code of the code of the covering operation of FIG. 6C. FIG. 6E shows the code generated after applying register coalescing and allocating to the code of FIG. 6D.

However, since the code of FIG. 6D generates the NCI 'sllmov' after the register coalescing operation, as shown in FIG. 6E, the code may be eliminated. Thus, a new mutual exclusion edge may be added to the CIG of FIG. 6A.

FIGS. 7A through 7E are diagrams for describing a case where a final code is generated, according to another embodiment of the present invention. FIGS. 7A through 7E are diagrams of a CIG, a labeling operation, a covering operation, a code at the covering operation, and a code generated after a register allocation operation, respectively, with respect to Example.

As described above, with referenced to a CIG of FIG. 7A, a new mutual exclusion edge is added between D and E in the CIG of FIG. 6A. Compatible CIC sets {B, E}, {B, F}, {D} and {G} are recomputed, among which the first two sets have maximum profit 1.5.

In a third iteration, after the labeling operation of FIG. 7B and the covering operation of FIG. 7C are performed, the code of FIG. 7D is generated. At the instruction selection operation, any set is selected. As shown in FIG. 7C, when {B, E} is selected, the code of FIG. 7D is generated.

At last, after the third iteration, as shown in FIG. 7E, a final code with minimum cost having the instruction 'mov' having no NCI is obtained, wherein the instruction 'mov' may be simultaneously eliminated.

In order to obtain the code having no NCI, the instruction selector 110 may generate the code of FIG. 7D.

Unfortunately, this is impossible since this means that the compiler has selected the CIC set {B, E} which has a less profit than the original choice, at the earlier CSS operation.

A policy of selecting the most compatible CIC set according to its profit at the CSS operation increases the chance of the selected CIC being selected successfully as a compound instruction. However, the policy does not guarantee the final code with minimum cost. This is because the NCI results from the register coalescing.

Whenever different compatible CICs are selected, the mixture or order of instructions in the code is changed, thus altering live ranges of temporaries. This implies that the colorability of each node of an interference graph may be known only after the CIC set is fixed.

Unless all CIC sets are tested for possible coalescing beforehand, it may not be determined which choice of CICs result in NICs in the final code at the CSS operation.

So far, the operation of the code generating apparatus according to the embodiments of the present invention has been described. The operation (function) of the code generating apparatus has the same technological feature as that of a method of generating a code, which will be described, and thus its details will not be repeated.

FIG. 8 is a flowchart of a method of generating a code, according to an embodiment of the present invention.

First, the instruction selector 110 selects a CIC from a node 'G' of the CIG (operation S11).

When it is assumed that sets of compatible CICs computed from 'G' of the CIG at the CSS operation is Ψ , the instruction selector 110 selects a CIC set ψ such that $\psi \in \Psi$ and $\text{profit}(\psi) \geq \text{profit}(\phi)$ for every $\phi \in \Psi$.

In addition, the instruction selector 110 evaluates costs of all split rules that are split from the complex rule for ψ .

The instruction selector 110 performs a labeling operation (operation S12) and a covering operation (operation S13), by using the split rules along with all simple rules.

After the covering operation (operation S13) is performed, the register allocator 120 performs register allocation and register coalescing, generates a code,

and then transmits the generated code to the NCI detector 130 and the final code generator 140 (operation S14).

The NCI detector 130 detects whether the generated code has any NCI generated at the register allocation operation (operation S15).

5 If the NCI detector 130 does not detect any NCI, the NCI detector 130 notifies the final code generator 140 about nonexistence of NCI, and allows the final code generator 140 to emit the final code (operation S17).

10 If the NCI detector 130 detects the NCI, the NCI detector 130 allows the instruction selector 110 to eliminate the CIC creating the NCI from the CIG, and to select a new CIC (operation S16).

In this case, the instruction selector 110 determines X belonging to ψ as a CIC corresponding to the NCI. If X is the only CIC of ψ , X is eliminated from the CIG. Otherwise, a mutual exclusion edge as a new interference edge is inserted into 'G' of the CIG (operation S16). In this case, every Y belongs to ψ (every $Y \in \psi$).

15 In operation S16, since at least one interference edge is inserted into the node 'G' of the CIG, or one CIC is eliminated from the node 'G', the number of iterations may be $O(k^2)$, in the worst case, where 'k' is the number of CICs in the CIG. Normally, the NCIs no longer appear at early iterations, the number of iterations is reduced to $O(k)$.

20 The most time consuming operations are operations S11 and S15. Thus, 'k' is asymptotically linear to 'n' (the number of nodes in IR trees), the overall instruction selection time is $O(n^3)$ on average.

25 Anyhow, as verified in this experiment, compilation times may be maintained in a reasonable time bound $O(n)$ even if compound instructions for code generation are additionally handled.

30 The invention can also be embodied as computer readable codes on a computer readable recording medium. The computer readable recording medium is any data storage device that can store data which can be thereafter read by a computer system. Examples of the computer readable recording medium include read-only memory (ROM), random-access memory (RAM), CD-ROMs, magnetic tapes, floppy disks, optical data storage devices, etc. The computer readable recording medium can also be distributed over network coupled computer systems so that the computer readable code is stored and executed in a distributed fashion.

According to one or more embodiment of the present invention, the quality of the final target code may be improved by preventing generation of the NIC by considering register coalescing used at a register allocation operation during the compound instruction selection.

5 While the present invention has been particularly shown and described with reference to exemplary embodiments thereof, it will be understood by those of ordinary skill in the art that various changes in form and details may be made therein without departing from the spirit and scope of the present invention as defined by the following claims.

What is claimed is:

1. An apparatus for generating an optimized code by eliminating nullified compound instruction during compound instruction selection, the apparatus comprising:
an instruction selector for selecting a compound instruction candidate set which can be executed in parallel from a candidate interference graph consisting of compound instruction candidates extracted from an intermediate representation tree;
a register allocator for generating a code to which a register is allocated by allocating and merging a register to be used in the compound instruction candidate selected by the instruction selector;
a nullified compound instruction detector for detecting a nullified compound instruction including a nullified move instruction in the code to which the register is allocated; and
a final code generator for emitting the code to which the register is allocated as a final code, when the nullified compound instruction is not detected,
wherein the nullified compound instruction detector eliminates the selected compound instruction candidate set, and allows the instruction selector to select a new compound instruction candidate set, when the nullified compound instruction is detected.
2. The apparatus of claim 1, wherein the nullified compound instruction detector adds a mutual exclusion edge implying mutual exclusion between nodes related to the selected compound instruction candidate in the candidate interference graph and allows the nodes connected by the mutual exclusion edge not to be selected together as the compound instruction candidate set.
3. The apparatus of claim 1, wherein the instruction selector selects the compound instruction candidate set with a maximum weight sum in the candidate interference graph.
4. A method of generating an optimized code by eliminating nullified compound instruction during compound instruction selection, the method comprising:
selecting a compound instruction candidate set which can be executable in parallel from a candidate interference graph consisting of compound instruction candidates extracted from an intermediate representation tree;
generating a code to which a register is allocated by allocating and merging a register to be used in the selected compound instruction candidate;
detecting a nullified compound instruction including a nullified move instruction in the code to which the register is allocated;
emitting the code to which the register is allocated as a final code, when the nullified compound instruction is not detected; and
eliminating the selected compound instruction candidate set from the candidate interference

graph, and returning back to the selecting of the compound instruction candidate set, when the nullified compound instruction is detected.

5. The method of claim 4, further comprising, prior to the generating of the code, performing labeling and covering on the intermediate representation tree.